

Overview of Programming with Java Lambdas & Streams

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

**Institute for Software
Integrated Systems**

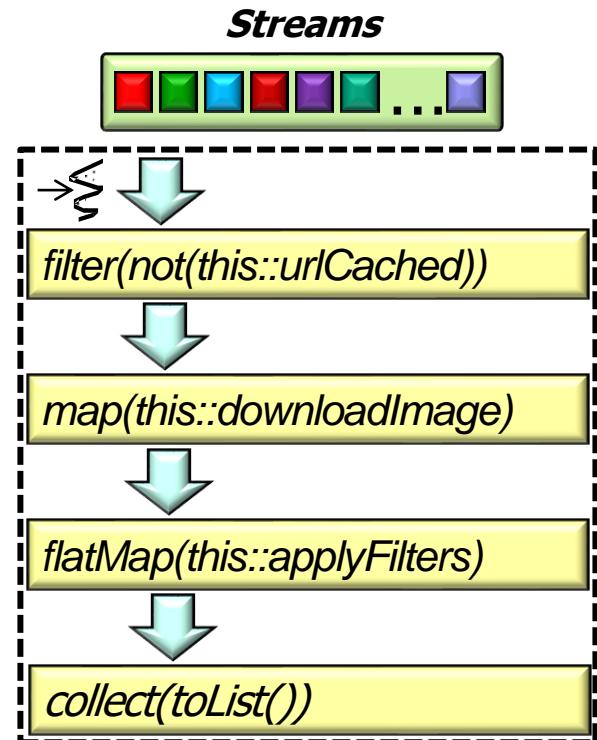
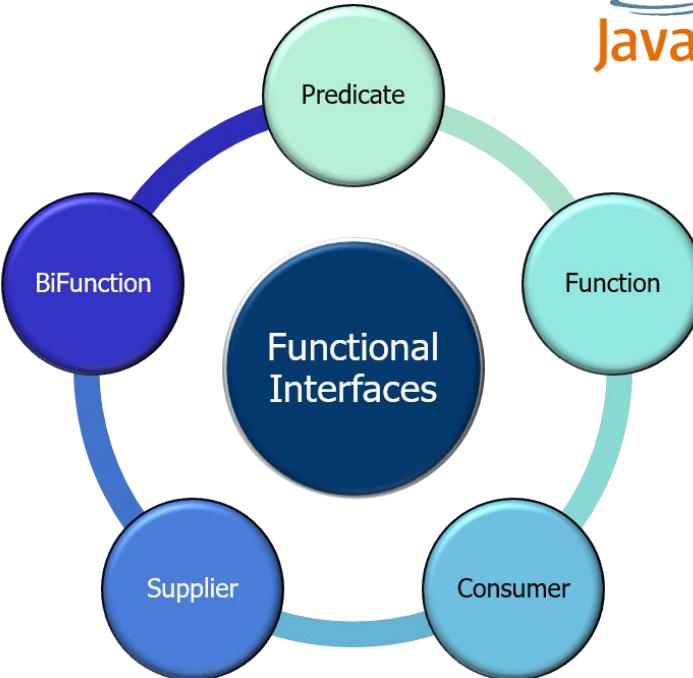
**Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Lesson

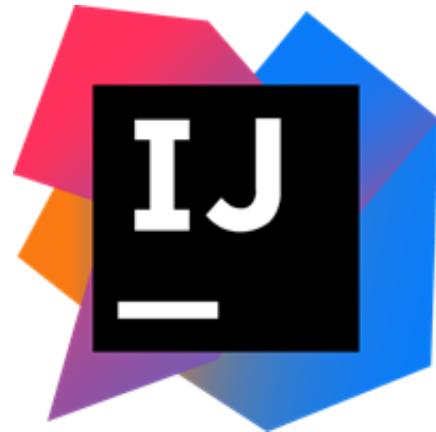
- Know what topics we'll cover

λ



Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java 8+ & relevant IDEs



Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java 8+ & relevant IDEs
- Be aware of other digital learning resources



Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java 8+ & relevant IDEs
- Be aware of other digital learning resources
- Be able to locate examples of Java 8+ programs

USE THE
SOURCE LIVED
LESSONS



Branch: master			New pull request	Create new file	Upload files	Find file	Clone or download
douglasraigschmidt	updates						Latest commit a67fd89 35 minutes ago
BarrierTaskGang	Updates						10 months ago
BuggyQueue	updates						a day ago
BusySynchronizedQueue	updates						4 months ago
DeadlockQueue	Refactored						3 years ago
ExpressionTree	Updates						10 months ago
Factorials	update						5 days ago
ImageStreamGang	updates						22 days ago
ImageTaskGangApplication	Updates						10 months ago
Java8	update						3 days ago
PalantirManagerApplication							7 months ago
PingPongApplication							10 months ago
PingPongWrong							3 years ago
SearchStreamForkJoin							35 minutes ago
SearchStreamGang							3 hours ago
SearchStreamSpliterator							37 minutes ago
SearchTaskGang							4 months ago
SimpleAtomicLong							2 years ago
SimpleBlockingQueue	Updates						4 months ago
SimpleSearchStream	update						6 days ago
ThreadJoinTest	updates						22 days ago
ThreadedDownloads	Updates						10 months ago
UserOrDaemonExecutor	Refactored						3 years ago
UserOrDaemonRunnable	Updates.						2 years ago
UserOrDaemonThread	Updates						10 months ago
UserThreadInterrupted	Update						2 years ago
.gitattributes	Committed.						3 years ago
.gitignore	Updates						10 months ago
README.md	updates						21 days ago

*We'll review many of
these examples, so
feel free to clone or
download this repo!*

See www.github.com/douglasraigschmidt/LiveLessons

Overview of this Course

Overview of this Course

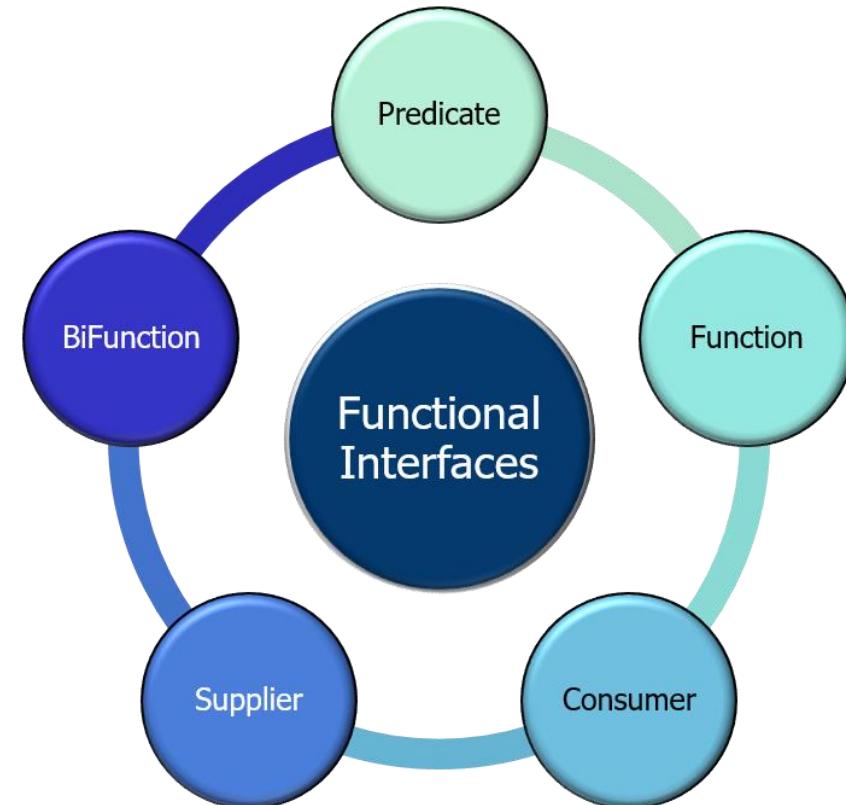
- We focus on using core Java functional programming features



Overview of this Course

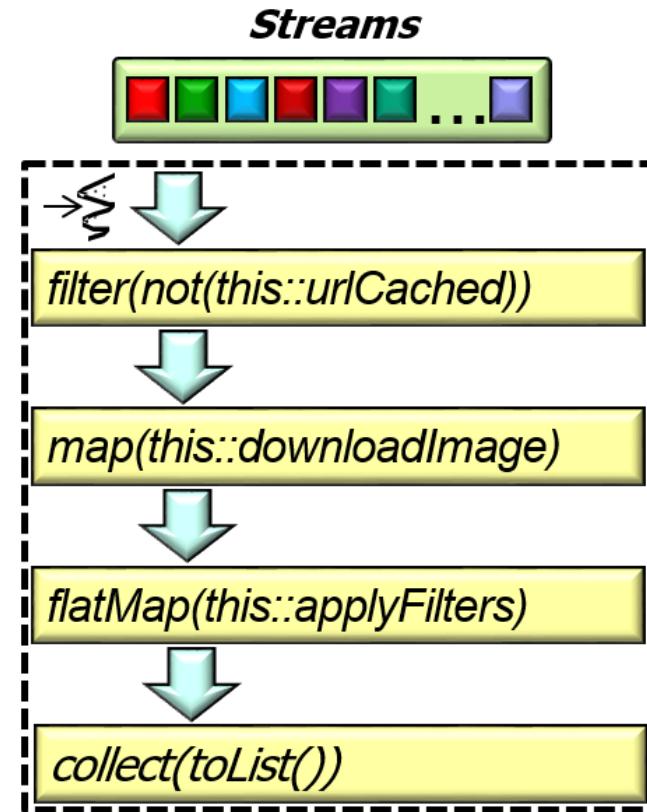
- We focus on using core Java functional programming features, e.g.
 - Lambda expressions, method references, & functional interfaces

λ



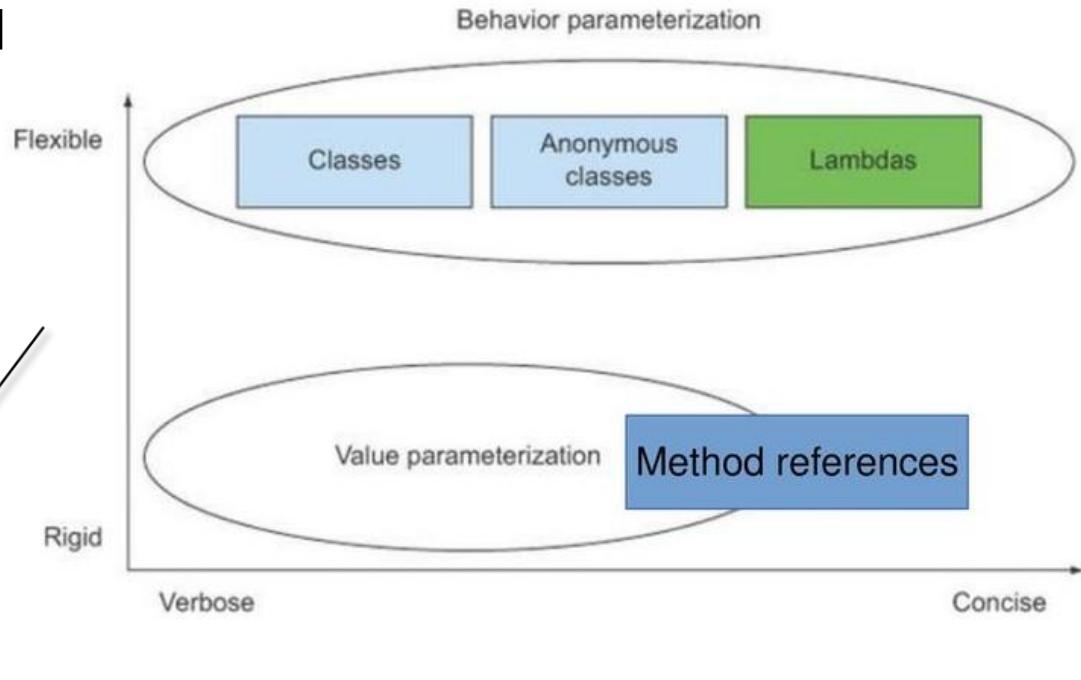
Overview of this Course

- We focus on using core Java functional programming features, e.g.
 - Lambda expressions, method references, & functional interfaces
 - The streams framework
 - Manipulate flows of elements declaratively via function composition



Overview of this Course

- We focus on using core Java functional programming features, e.g.
 - Lambda expressions, method references, & functional interfaces
 - The streams framework

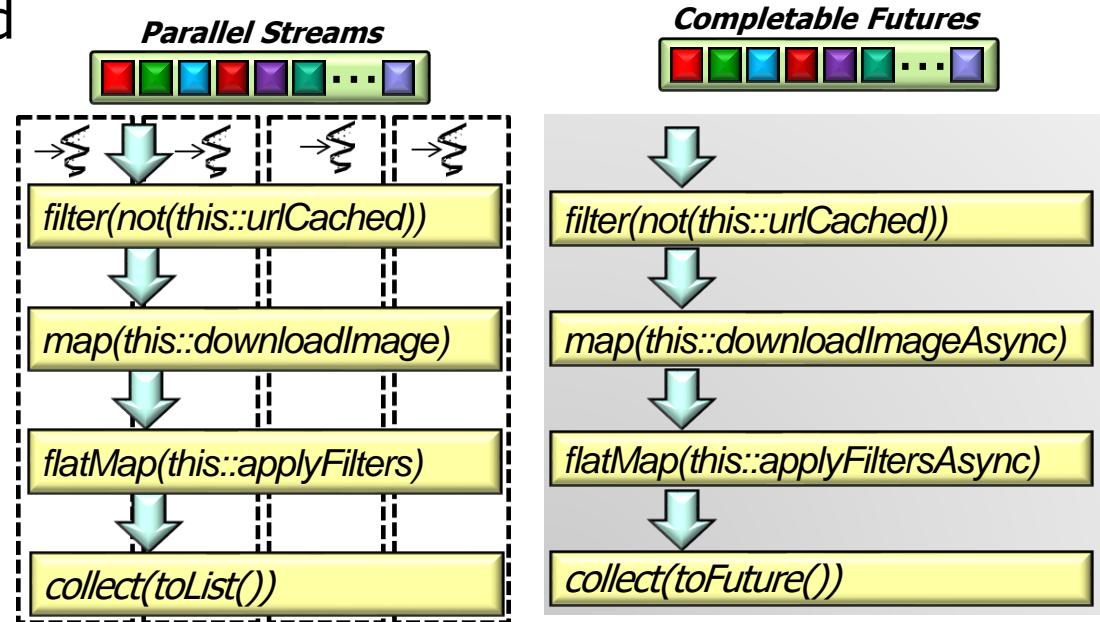


These functional programming features support concise "behavior parameterization"

See blog.indrek.io/articles/java-8-behavior-parameterization

Overview of this Course

- We focus on using core Java functional programming features, e.g.
 - Lambda expressions, method references, & functional interfaces
 - The streams framework



These features are the foundation for Java's concurrency/parallelism frameworks

Overview of this Course

- All Java programming examples we cover are available on github

douglascraigschmidt	updates	Latest commit a67fd89 38 minutes ago
BarrierTaskGang	Updates	10 months ago
BuggyQueue	updates	a day ago
BusySynchronizedQueue	updates	4 months ago
DeadlockQueue	Refactored	3 years ago
ExpressionTree	Updates	10 months ago
Factorials	update	5 days ago
ImageStreamGang	updates	22 days ago
ImageTaskGangApplication	Updates	10 months ago
Java8	update	3 days ago
PalantiriManagerApplication	Updates	7 months ago
PingPongApplication	Updates	10 months ago
PingPongWrong	Refactored	3 years ago
SearchStreamForkJoin	updates	38 minutes ago

See www.github.com/douglascraigschmidt/LiveLessons

Accessing Java 8+ Features & Functionality

Accessing Java 8+ Features & Functionality

- The Java 8+ runtime environment (JRE) supports modern Java features

Overview Downloads Documentation Community Technologies Training

Java SE Downloads

 DOWNLOAD  DOWNLOAD

Java Platform (JDK) 8u101 / 8u102 NetBeans with JDK 8

Java Platform, Standard Edition

Java SE 8u101 / 8u102
Java SE 8u101 includes important security fixes. Oracle strongly recommends that all Java SE 8 users upgrade to this release. Java SE 8u102 is a patch-set update, including all of 8u101 plus additional features (described in the release notes).
[Learn more](#)

- Installation Instructions
- Release Notes
- Oracle License
- Java SE Products
- Third Party Licenses
- Certified System Configurations
- Readme Files
 - JDK ReadMe
 - JRE ReadMe

JDK DOWNLOAD
Server JRE DOWNLOAD
JRE DOWNLOAD



See docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html

Accessing Java 8+ Features & Functionality

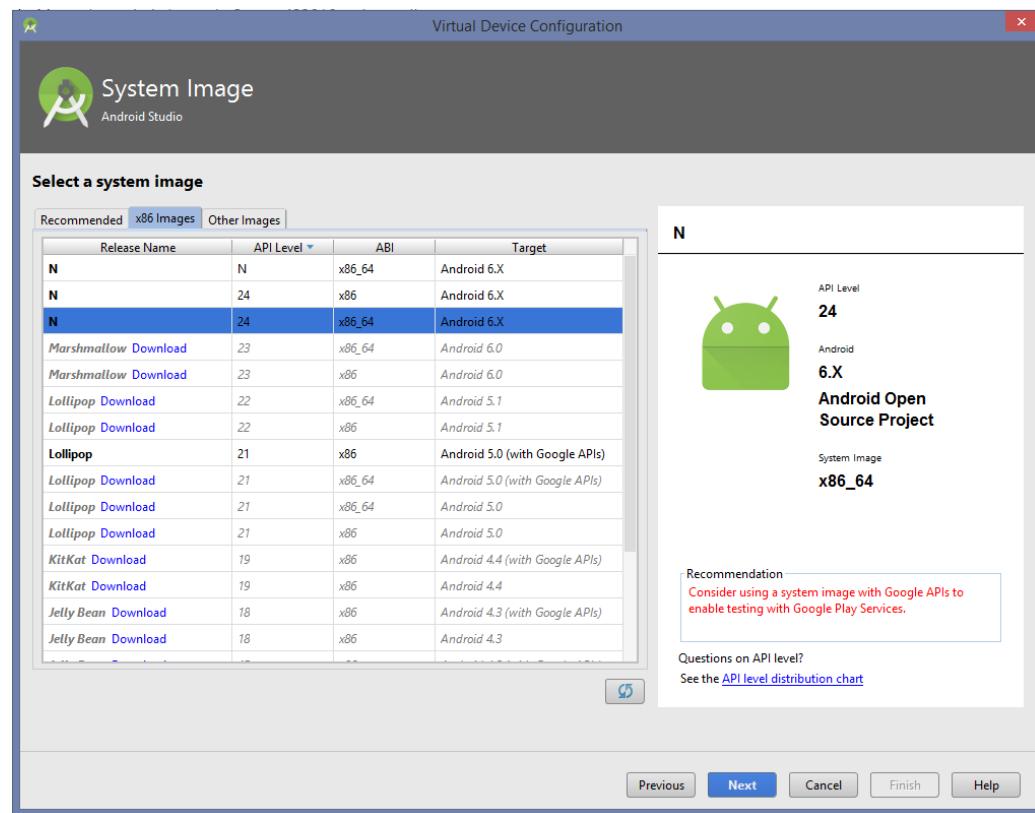
- The Java 8+ runtime environment (JRE) supports modern Java features, e.g.
 - IntelliJ & Eclipse are popular Java IDEs



See www.eclipse.org/downloads & www.jetbrains.com/idea/download

Accessing Java 8+ Features & Functionality

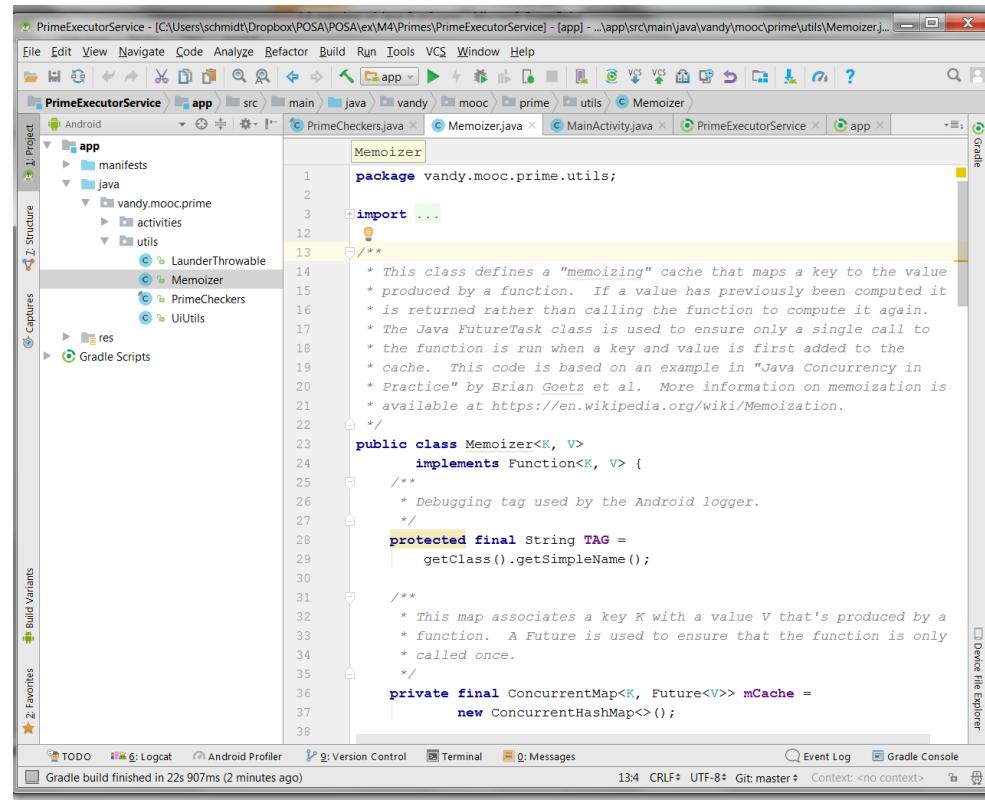
- The Java 8+ runtime environment (JRE) supports modern Java features, e.g.
 - IntelliJ & Eclipse are popular Java IDEs
 - Most Java 8 features are supported by Android API level 24 (& beyond)



See android-developers.googleblog.com/2017/03/future-of-java-8-language-feature.html

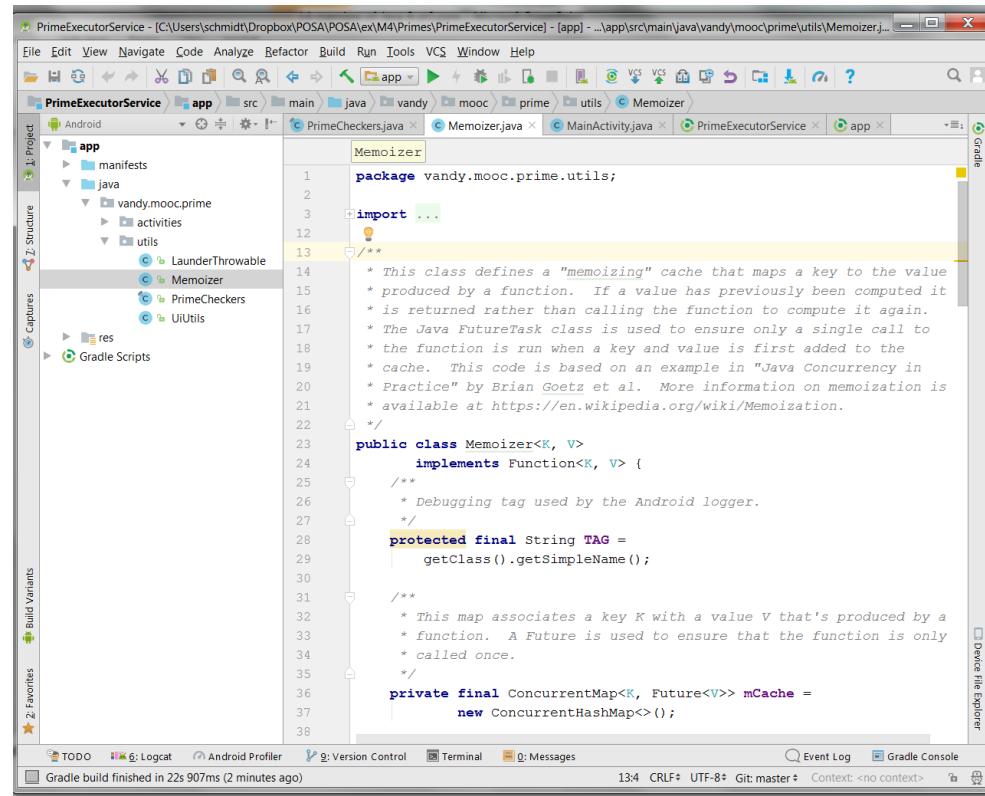
Accessing Java 8+ Features & Functionality

- The Java 8+ runtime environment (JRE) supports modern Java features, e.g.
 - IntelliJ & Eclipse are popular Java IDEs
 - Most Java 8 features are supported by Android API level 24 (& beyond)
 - A subset of Java 8 features are available in earlier Android releases, as well



Accessing Java 8+ Features & Functionality

- The Java 8+ runtime environment (JRE) supports modern Java features, e.g.
 - IntelliJ & Eclipse are popular Java IDEs
 - Most Java 8 features are supported by Android API level 24 (& beyond)
 - A subset of Java 8 features are available in earlier Android releases, as well
 - Make sure to get Android Studio 3.x or later!



See developer.android.com/studio/preview/features/java8-support.html

Accessing Java 8+ Features & Functionality

- Java 8+ source code is available online
 - For browsing searchcode.com
 - For downloading
jdk8.java.net/download.html



The screenshot shows the Java.net website with the URL <http://jdk8.java.net/>. The page title is "JDK 8 Project" with the subtitle "Building the next generation of the JDK platform". On the left, there's a sidebar with a "JDK 8" menu containing links for "Downloads", "Feedback Forum", "OpenJDK", and "Planet JDK". The main content area features a section titled "JDK 8 snapshot builds" with a list of links: "Download 8u40 early access snapshot builds", "Source code (instructions)", "Official Java SE 8 Reference Implementations", "Early Access Build Test Results (instructions)". Below this is a "Feedback" section with instructions for reporting bugs and using the project forum.

JDK 8 Project
Building the next generation of the JDK platform

JDK 8 snapshot builds

- Download 8u40 early access snapshot builds
- Source code (instructions)
- Official Java SE 8 Reference Implementations
- Early Access Build Test Results (instructions)

Feedback

Please use the [Project Feedback](#) forum if you have suggestions for or encounter issues using JDK 8.

If you find bugs in a release, please submit them using the usual [Java SE bug reporting channels](#), not with the Issue tracker accompanying this project. Be sure to include complete version information from the output of the `java -version` command.

Other Digital Learning Resources

Other Digital Learning Resources

- There are several related Live Training courses coming soon

Scalable Programming with Java 8 Parallel Streams



DOUGLAS SCHMIDT



Reactive Programming with Java 8 Completable Futures



DOUGLAS SCHMIDT



Core Java Synchronizers



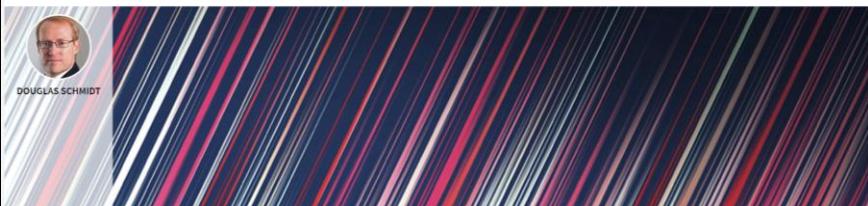
DOUGLAS SCHMIDT



Design Patterns in Java



DOUGLAS SCHMIDT



Scalable Concurrency with the Java Executor Framework



DOUGLAS SCHMIDT



Scalable Reactive Programming with Java

Topic: Software Development



DOUGLAS SCHMIDT



See www.dre.vanderbilt.edu/~schmidt/DigitalLearning

Other Digital Learning Resources

- Examples not covered here are covered in my other LiveLessons courses

The image displays three digital learning resource cards side-by-side.

- Parallel Function Programming with Java** by Douglas C. Schmidt. This card has an orange background. On the left, there is a sidebar with a blue header "douglasraigschmidt updates" and a list of links: BarrierTaskGang, BuggyQueue, BusySynchronizedQueue, DeadlockQueue, ExpressionTree, and Factorials. The main content area features the title "Parallel Function Programming with Java" and author "Douglas C. Schmidt". Below the author's name is the Pearson logo (a blue circle with a white letter P). To the right of the main content is a dark grey sidebar containing a screenshot of a computer screen showing code and a small video thumbnail of a man in a maroon shirt holding a laptop.
- Design Patterns in Java LiveLessons** by Douglas C. Schmidt. This card has an orange background. It features the title "Design Patterns in Java LiveLessons" and author "Douglas C. Schmidt". At the bottom left is a sidebar with links: PingPongWrong and SearchStreamForkJoin. At the bottom right is the Addison-Wesley logo (a red triangle with a white star). To the right of the main content is a dark grey sidebar containing a screenshot of a computer screen showing code and a small video thumbnail of a man in a maroon shirt holding a laptop.
- Java Concurrency LiveLessons 2nd Edition** by Doug Schmidt. This card has an orange background. It features the title "Java Concurrency LiveLessons 2nd Edition" and author "Doug Schmidt". To the right of the main content is a dark grey sidebar containing a screenshot of a computer screen showing code and a small video thumbnail of a man in a maroon shirt holding a laptop.

See www.dre.vanderbilt.edu/~schmidt/LiveLessons

Other Digital Learning Resources

- There's also a Facebook group dedicated to discussing Java-related topics

The image shows a screenshot of a Facebook group page. The group is titled "Java Concurrency LiveLessons" and is described as "Concurrent and Parallel Programming in Java". It has 1.9K members. The page features a large orange banner with the text "Java Concurrency LiveLessons" and a play button icon. To the right of the banner, there is a video player showing a man in a maroon shirt sitting at a desk with a laptop. Below the video player is an "Edit" button. At the bottom of the page, there is a row of user profile pictures and a blue "+ Invite" button.

See www.facebook.com/groups/1532487797024074

Other Digital Learning Resources

- See my website for many more videos & screencasts related to programming with Java & associated topics



Digital Learning Offerings

Douglas C. Schmidt (d.schmidt@vanderbilt.edu)
Associate Chair of [Computer Science and Engineering](#),
[Professor](#) of Computer Science, and Senior Researcher
in the [Institute for Software Integrated Systems \(ISIS\)](#)
at [Vanderbilt University](#)



O'Reilly LiveTraining Courses

- Programming with Java 8 Lambdas and Streams
 - [January 9th, 2018, 9:00am-12:00pm central time](#)
 - [February 1st, 2018, 9:00am-12:00pm central time](#)
 - March 1st, 2018, 9:00am-12:00pm central time
- Scalable Programming with Java 8 Parallel Streams
 - [January 10th, 2018, 11:00am-3:00pm central time](#)
 - February 6th, 2018, 11:00am-3:00pm central time
 - March 6th, 2018, 11:00am-3:00pm central time
- Reactive Programming with Java 8 Completable Futures
 - [January 12th, 2018, 10:00am-1:00pm central time](#)
 - [February 13th, 2018, 10:00am-2:00pm central time](#)
 - March 13th, 2018, 10:00am-2:00pm central time

Pearson LiveLessons Courses

- [Java Concurrency](#)
- [Design Patterns in Java](#)

Coursera MOOCs

- [Android App Development](#) Coursera Specialization
- [Pattern-Oriented Software Architecture \(POSA\)](#)

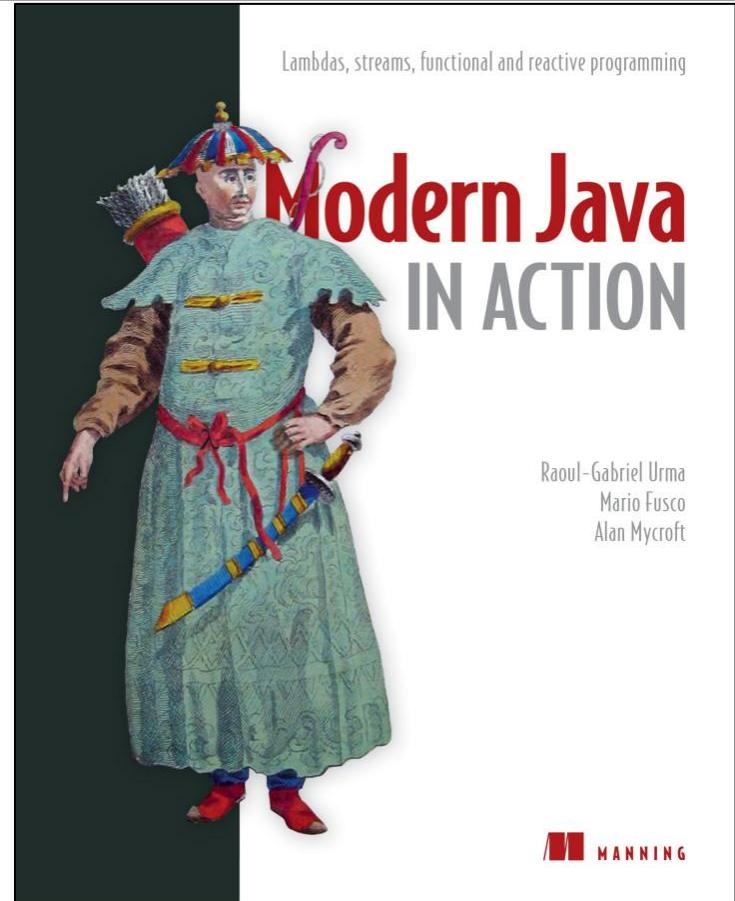
Vanderbilt University Courses

- [Playlist](#) from my [YouTube Channel](#) videos from [CS 891: Introduction to Concurrent and Parallel Java Programming with Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 892: Concurrent Java Programming with Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 251: Intermediate Software Design with Java](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 282: Concurrent Java Network Programming in Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 251: Intermediate Software Design with C++](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 282: Systems Programming for Android](#)

See www.dre.vanderbilt.edu/~schmidt/DigitalLearning

Other Digital Learning Resources

- Another excellent source of material to read is the book *Modern Java in Action*



See www.manning.com/books/modern-java-in-action

Other Digital Learning Resources

- Another excellent source of material to read is the book *Modern Java in Action*
 - There are also good online articles based on this book

Processing Data with Java SE 8 Streams, Part 1

by Raoul-Gabriel Urma

Use stream operations to express sophisticated data processing queries.

What would you do without collections? Nearly every Java application *makes and processes* collections. They are fundamental to many programming tasks: they let you group and process data. For example, you might want to create a collection of banking transactions to represent a customer's statement. Then, you might want to process the whole collection to find out how much money the customer spent. Despite their importance, processing collections is far from perfect in Java.

First, typical processing patterns on collections are similar to SQL-like operations such as "finding" (for example, find the transaction with highest value) or "grouping" (for example, group all transactions related to grocery shopping). Most databases let you specify such operations declaratively. For example, the following SQL query lets you find the transaction ID with the highest value: "SELECT id, MAX(value) from transactions".

As you can see, we don't need to implement *how* to calculate the maximum value (for example, using loops and a variable to track the highest value). We only express *what* we expect. This basic idea means that you need to worry less about how to explicitly implement such queries—it is handled for you. Why can't we do something similar with collections? How many times do you find yourself reimplementing these operations using loops over and over again?

Second, how can we process really large collections efficiently? Ideally, to speed up the processing, you want to leverage multicore architectures. However, writing parallel code is hard and error-prone.

Java SE 8 to the rescue! The Java API designers are updating the API with a new abstraction called *Stream* that lets you process data in a declarative way. Furthermore, streams can leverage multi-core architectures without you having to write a single line of multithread code. Sounds good, doesn't it? That's what this series of articles will explore.

Before we explore in detail what you can do with streams, let's take a look at an example so you have a sense of the new programming style with Java SE 8 streams. Let's say we need to find all transactions of type *grocery* and return a list of transaction IDs sorted in decreasing order of transaction value. In Java SE 7, we'd do that as shown in Listing 1. In Java SE 8, we'd do it as shown in Listing 2.

```
List<Transaction> groceryTransactions = new ArrayList<>();
for(Transaction t: transactions){
    if(t.getType() == Transaction.GROCERY){
        groceryTransactions.add(t);
    }
}
Collections.sort(groceryTransactions, new Comparator<>(){
    public int compare(Transaction t1, Transaction t2){
        return t2.getValue().compareTo(t1.getValue());
    }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
    transactionIds.add(t.getId());
}
```



Originally published in the March/April 2014 issue of *Java Magazine*. Subscribe today.

Here's a mind-blowing idea:
these two operations can produce elements "forever."

Other Digital Learning Resources

- Another excellent source of material to read is the book *Modern Java in Action*
 - There are also good online articles based on this book

Part 2: Processing Data with Java SE 8 Streams

by Raoul-Gabriel Urma

Published May 2014

Combine advanced operations of the Stream API to express rich data processing queries.

In the first part of this series, you saw that streams let you process collections with database-like operations. As a refresher, the example in Listing 1 shows how to sum the values of only expensive transactions using the Stream API. We set up a pipeline of operations consisting of intermediate operations (`filter`, `map`) and a terminal operation (`reduce`), as illustrated in Figure 1.

```
int sumExpensive =  
    transactions.stream()  
        .filter(t -> t.getValue() > 1000)  
        .map(Transaction::getValue)  
        .reduce(0, Integer::sum);
```

Listing 1

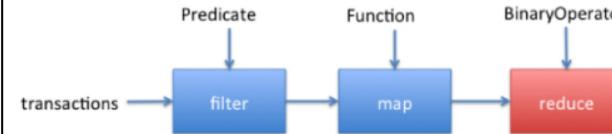


Figure 1

However, the first part of this series didn't investigate two operations:

- `flatMap`: An intermediate operation that lets you combine a "map" and a "flatten" operation
- `collect`: A terminal operation that takes as an argument various recipes (called collectors) for accumulating the elements of a stream into a summary result

These two operations are useful for expressing more-complex queries. For instance, you can combine `flatMap` and `collect` to produce a Map representing the number of occurrences of each character that appears in a stream of words, as shown in Listing 2. Don't worry if this code seems overwhelming at first. The purpose of this article is to explain and explore these two operations in more detail.

```
import static java.util.function.Function.identity;  
import static java.util.stream.Collectors.*;  
  
Stream<String> words = Stream.of("Java", "Magazine", "is",  
    "the", "best");  
  
Map<String, Long> letterToCount =  
    words.map(w -> w.split(""))  
        .flatMap(Arrays::stream)  
        .collect(groupingBy(identity(), counting()));
```



Originally published in the
May/June 2014 issue of *Java Magazine*. Subscribe today.

End of Course Overview

Overview of Java's Supported Programming Paradigms

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

**Institute for Software
Integrated Systems**

**Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Lesson

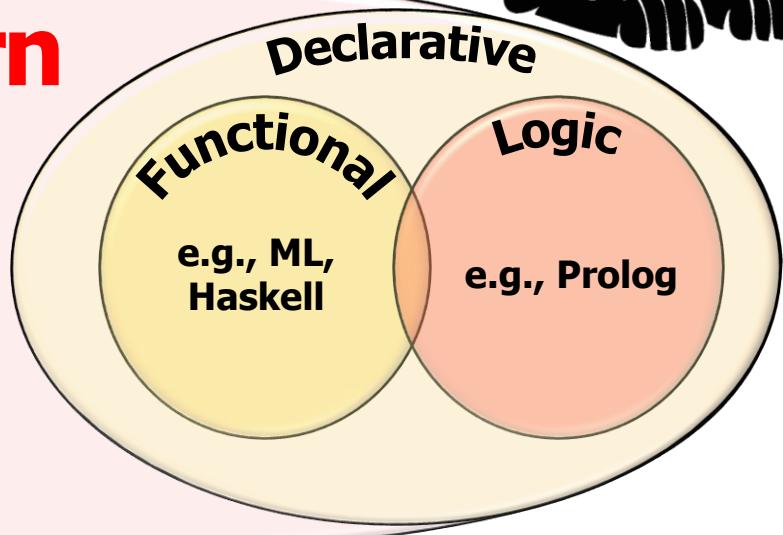
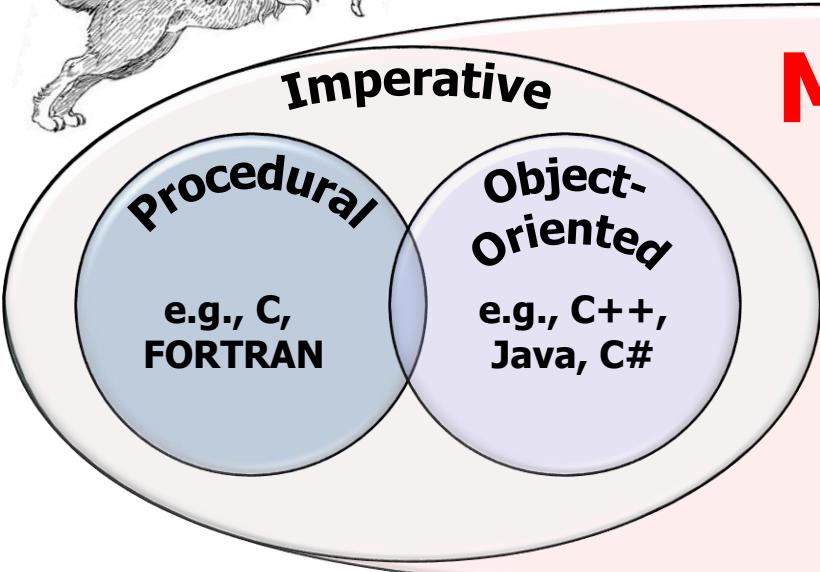
- Recognize the two programming paradigms supported by modern Java



Modern Java is a "hybrid" that combines object-oriented & functional paradigms

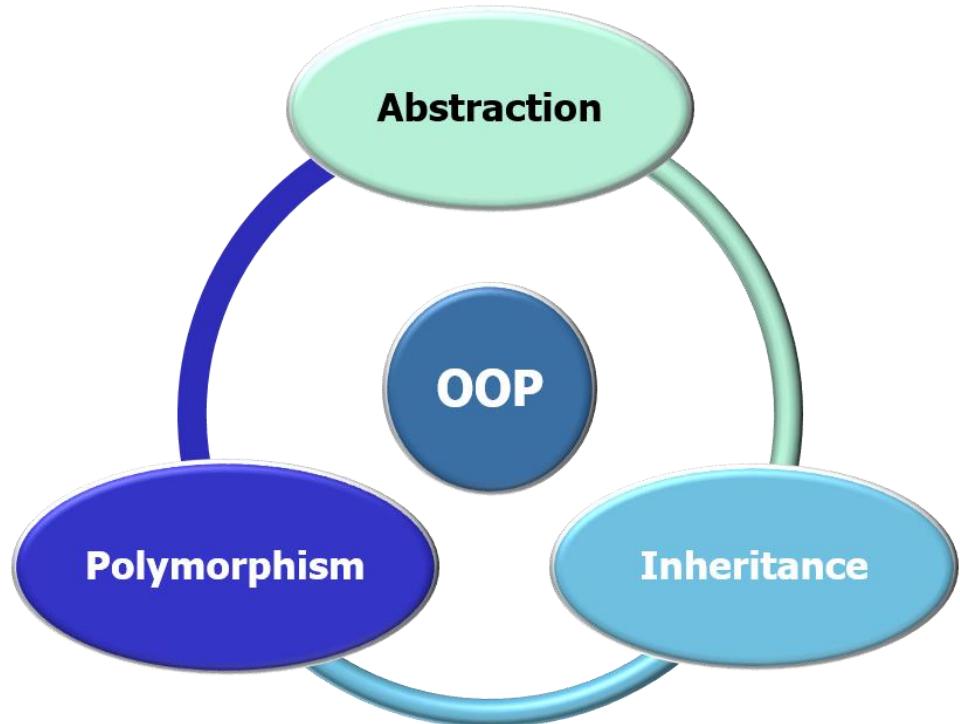


Modern Java



Learning Objectives in this Lesson

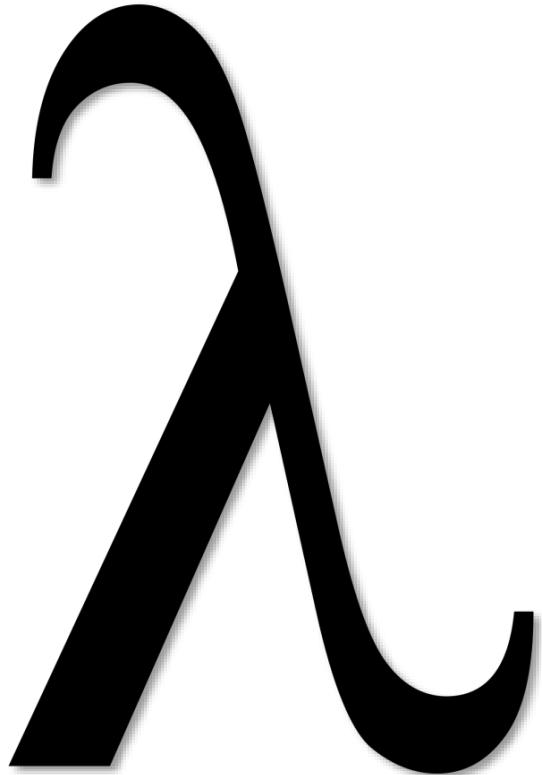
- Recognize the two programming paradigms supported by modern Java
 - Object-oriented programming



Java has supported object-oriented programming from the very beginning!

Learning Objectives in this Lesson

- Recognize the two programming paradigms supported by modern Java
 - Object-oriented programming
 - Functional programming



Naturally, these paradigms are also supported in versions above & beyond Java 8!

Learning Objectives in this Lesson

- Recognize the two programming paradigms supported by modern Java
 - Object-oriented programming
 - Functional programming

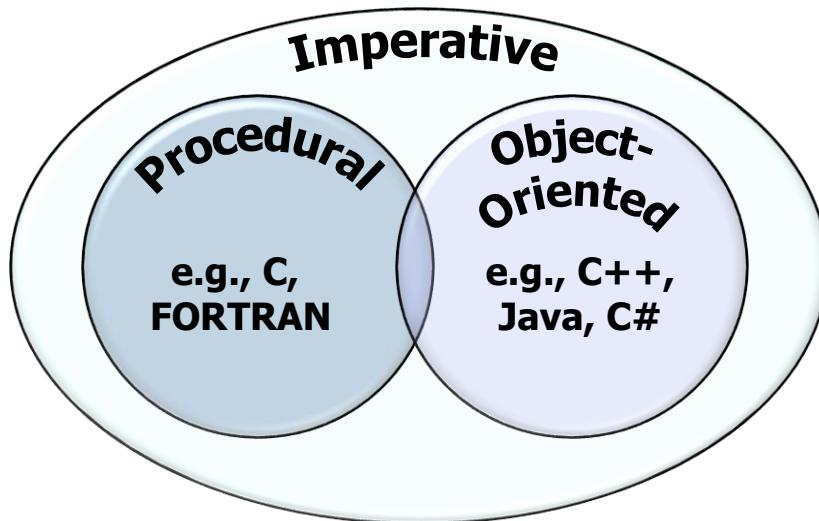


We show some modern Java code fragments that we'll cover in more detail later

Overview of the Imperative Programming Paradigm

Overview of the Imperative Programming Paradigm

- Object-oriented programming is an “imperative” paradigm

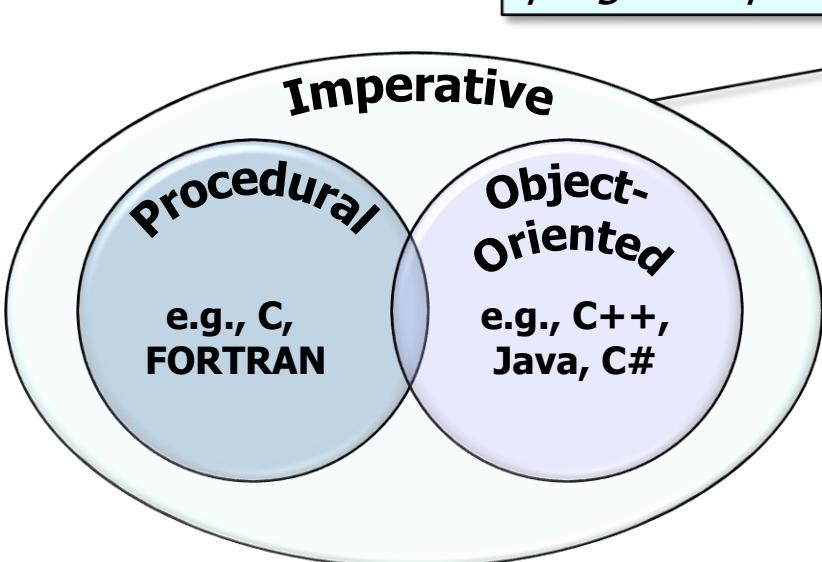


See en.wikipedia.org/wiki/Imperative_programming

Overview of the Imperative Programming Paradigm

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

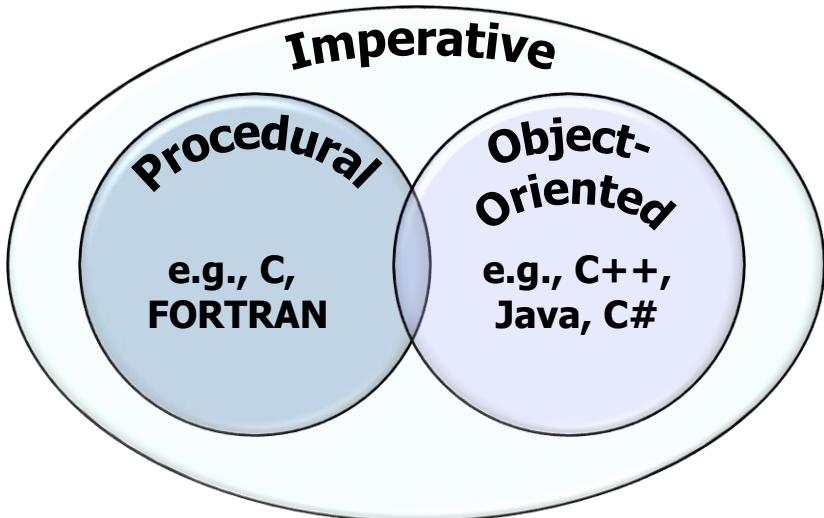
Imperative programming focuses on describing how a program operates via statements that change its state



Overview of the Imperative Programming Paradigm

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    List<String> res =  
        new ArrayList<>();  
    for (String line : lines)  
        if (!omit.equals(line))  
            res.add(line);  
  
    return res;  
}
```

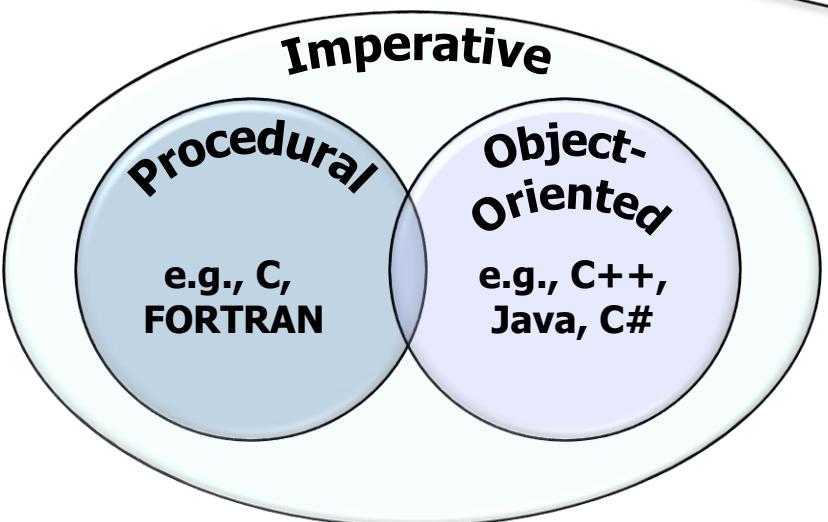


Imperatively remove a given string from a list of strings

Overview of the Imperative Programming Paradigm

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

Create an empty list to hold results

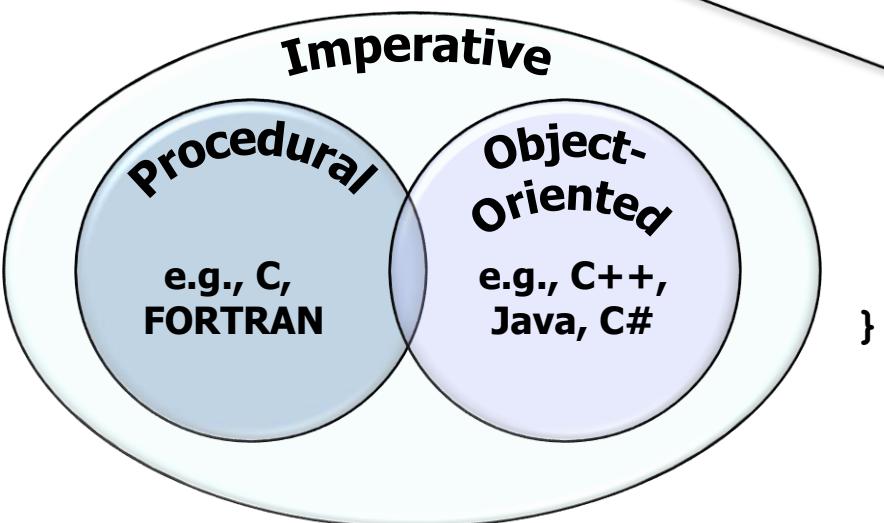


```
List<String> zap(List<String> lines,  
                  String omit) {  
    List<String> res =  
        new ArrayList<>();  
    for (String line : lines)  
        if (!omit.equals(line))  
            res.add(line);  
    return res;  
}
```

Overview of the Imperative Programming Paradigm

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

Iterate sequentially through each line



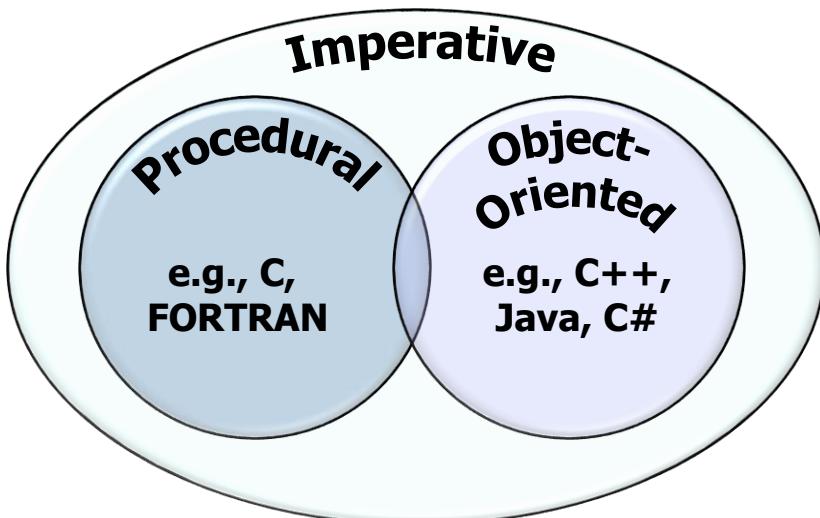
```
List<String> zap(List<String> lines,  
                  String omit) {  
    List<String> res =  
        new ArrayList<>();  
    for (String line : lines)  
        if (!omit.equals(line))  
            res.add(line);  
    return res;  
}
```

Overview of the Imperative Programming Paradigm

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    List<String> res =  
        new ArrayList<>();  
    for (String line : lines)  
        if (!omit.equals(line))  
            res.add(line);  
    return res;  
}
```

*Only add lines that don't
match the omit string*

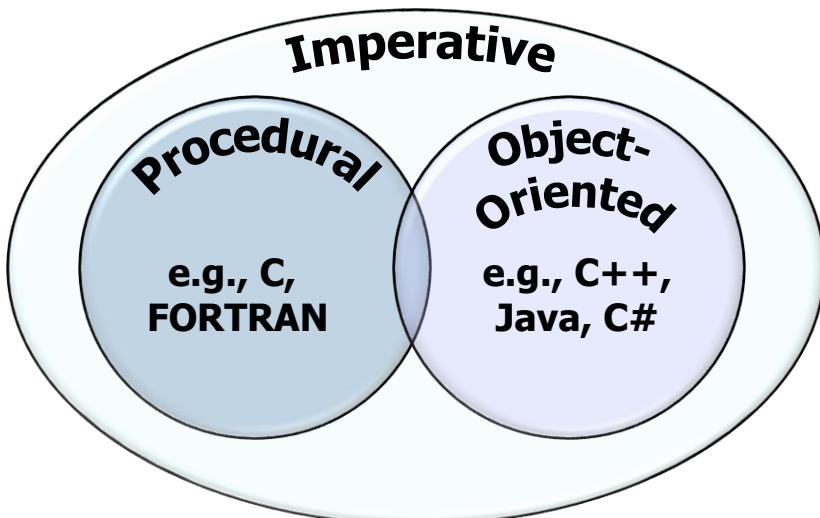


Overview of the Imperative Programming Paradigm

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

```
List<String> zap(List<String> lines,  
                  String omit) {  
    List<String> res =  
        new ArrayList<>();  
    for (String line : lines)  
        if (!omit.equals(line))  
            res.add(line);  
    return res;  
}
```

Return the list of non-matching lines

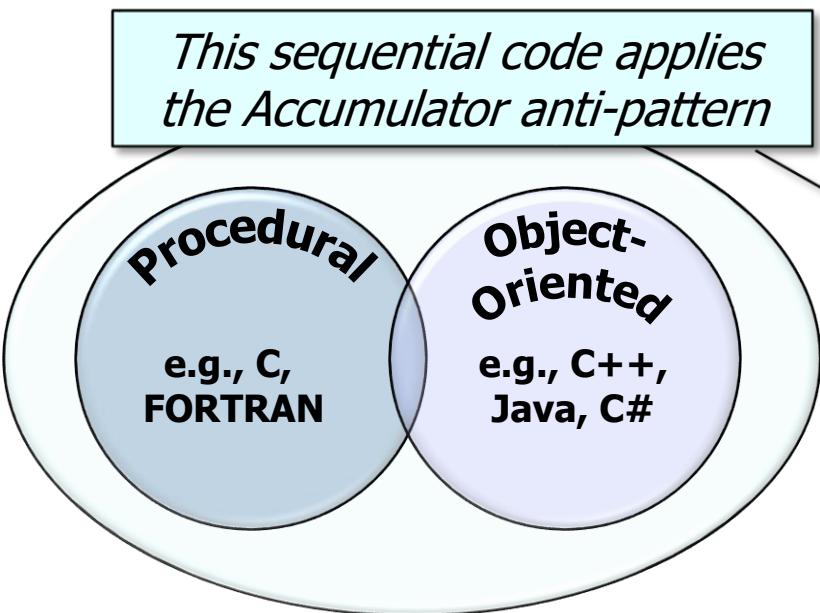


Overview of the Imperative Programming Paradigm

- Object-oriented programming is an “imperative” paradigm
 - e.g., a program consists of commands for the computer to perform

This sequential code applies the Accumulator anti-pattern

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    List<String> res =  
        new ArrayList<>();  
    for (String line : lines)  
        if (!omit.equals(line))  
            res.add(line);  
  
    return res;  
}
```

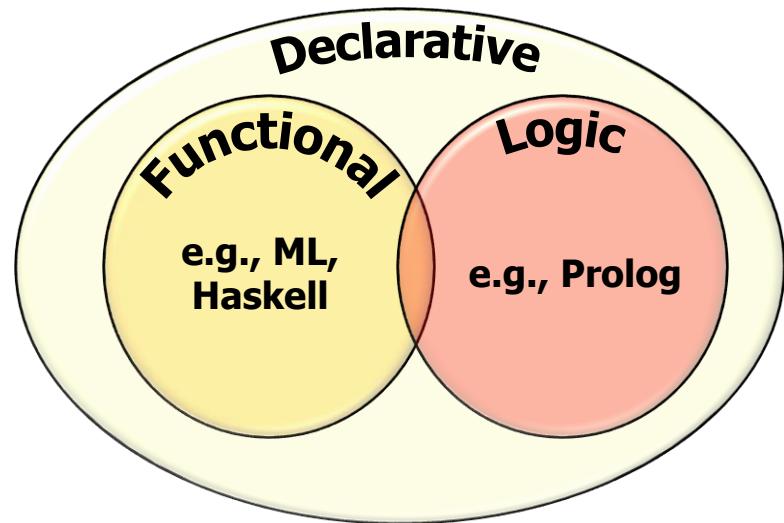


See developer.ibm.com/articles/j-java-streams-2-brian-goetz

Overview of the Declarative Programming Paradigm

Overview of the Declarative Programming Paradigm

- Functional programming is a “declarative” paradigm

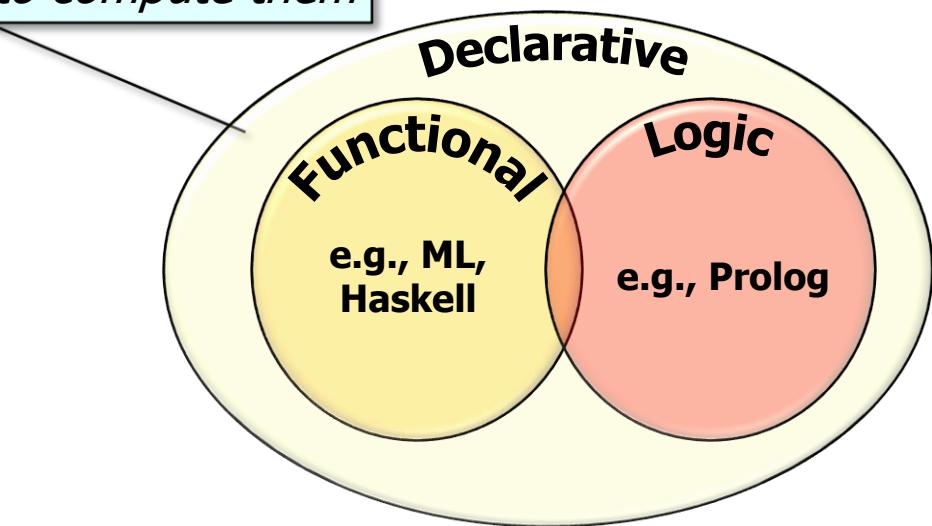
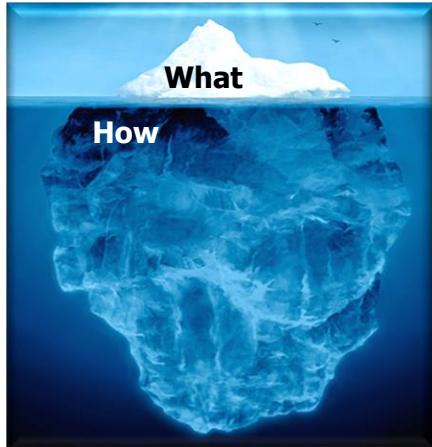


See en.wikipedia.org/wiki/Declarative_programming

Overview of the Declarative Programming Paradigm

- Functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

Declarative programming focuses on “what” computations should be performed, instead of on “how” to compute them

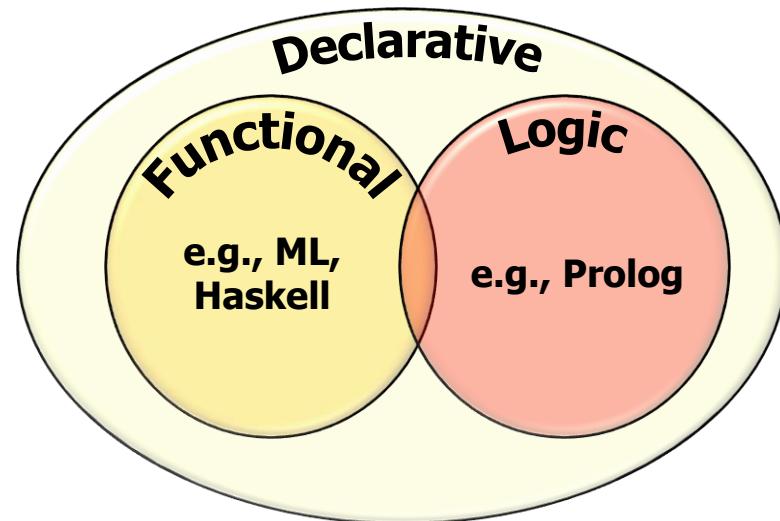


Overview of the Declarative Programming Paradigm

- Functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .stream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```

Declaratively remove a given string from a list of strings

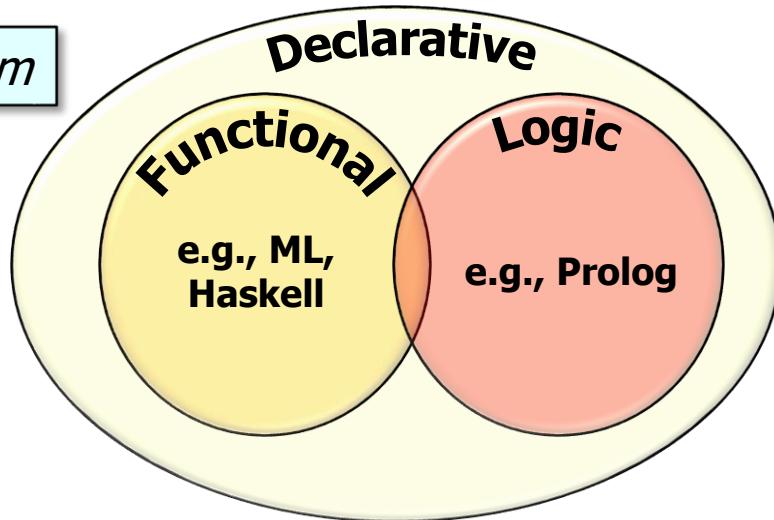


Overview of the Declarative Programming Paradigm

- Functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .stream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```

Convert list into a stream



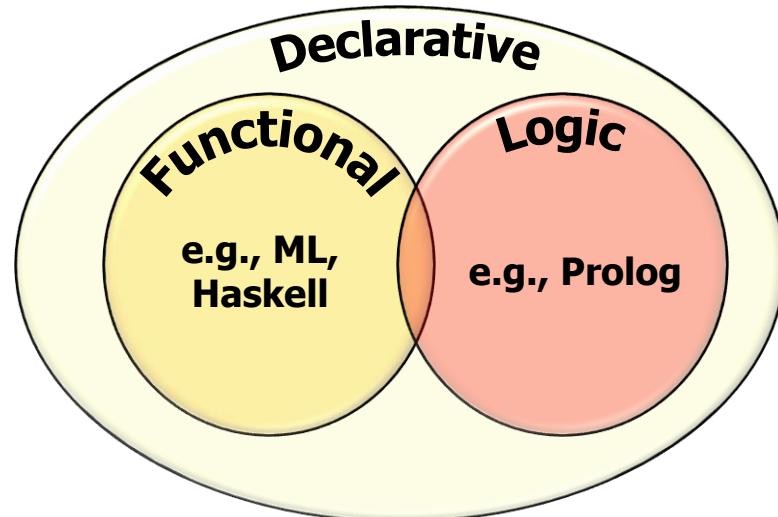
See docs.oracle.com/javase/tutorial/collections/streams

Overview of the Declarative Programming Paradigm

- Functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .stream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```

*Remove any line in the stream
that matches the `omit` param*

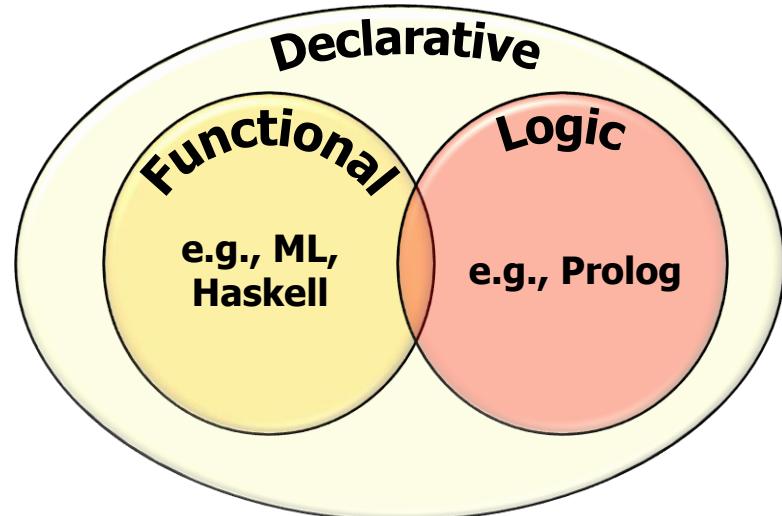


Overview of the Declarative Programming Paradigm

- Functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .stream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```

*Collect all non-matching lines
into a list & return it the caller*

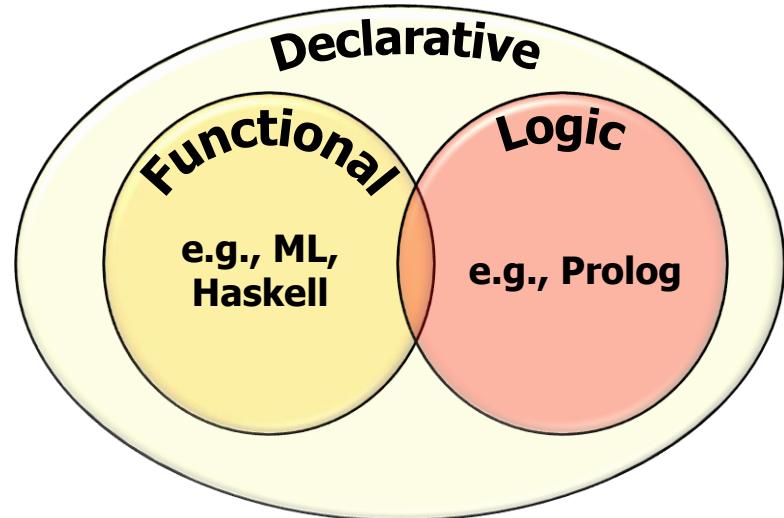


Overview of the Declarative Programming Paradigm

- Functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .stream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```

*Note “fluent” programming style
with cascading method calls*



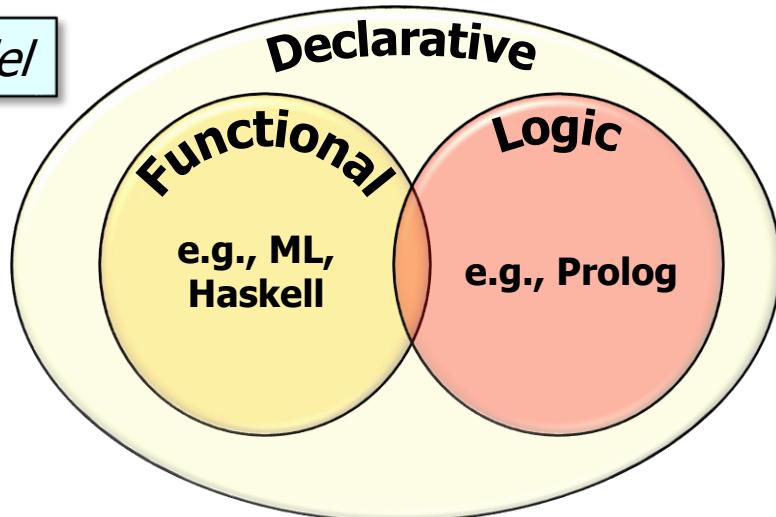
Overview of the Declarative Programming Paradigm

- Functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .parallelStream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```



Filter in parallel



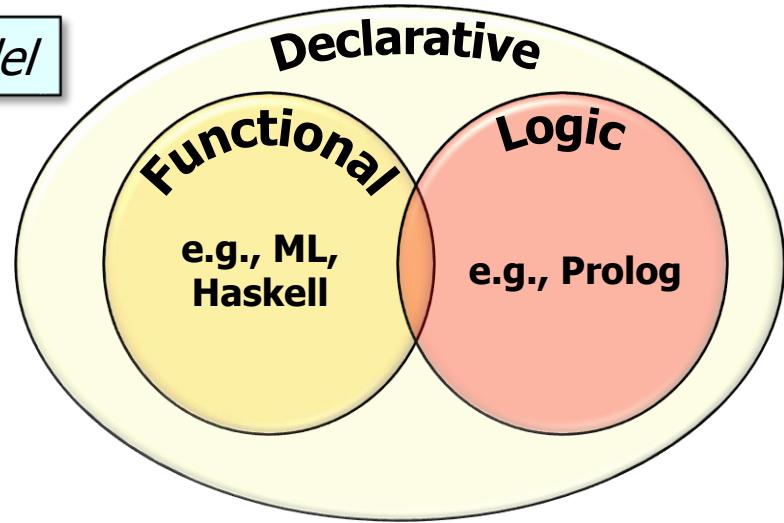
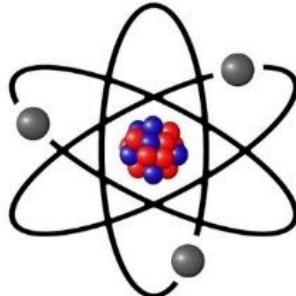
See docs.oracle.com/javase/tutorial/collections/stream/parallelism.html

Overview of the Declarative Programming Paradigm

- Functional programming is a “declarative” paradigm
 - e.g., a program expresses computational logic *without* describing control flow or explicit algorithmic steps

```
List<String> zap(List<String> lines,  
                  String omit) {  
  
    return lines  
        .parallelStream()  
        .filter(not(omit::equals))  
        .collect(toList());  
}
```

Filter in parallel



Code was parallelized with minuscule changes since it's declarative & stateless!

Summary of the Programming Paradigms in Modern Java

Summary of the Programming Paradigms in Modern Java

- Summary of these two paradigms

The screenshot shows a Twitter post from Michael Feathers (@mfeathers) dated Nov 3, 2010. The tweet reads: "OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts." This text is highlighted with a red box. Below the tweet, there are engagement metrics: 8 replies, 457 retweets, and 438 likes. The post is part of a thread under the heading "Replies". A reply from Kenji Hiranabe (@hiranabe) dated Jul 22, 2014, is shown, replying to @mfeathers. The reply text is in Japanese and translates to: "オブジェクト指向が変化をカプセル化してコードの理解性を上げたのに対し、関数型は変化を最小化してコードの理解性を上げた。TRT "@mfeathers: OO makes code understandable by encapsulating moving..."". The Twitter interface includes a search bar, login and sign-up buttons, and sections for "New to Twitter?" and "Relevant people".

Michael Feathers @mfeathers · Nov 3, 2010

OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

8 457 438

Replies

Kenji Hiranabe @hiranabe · Jul 22, 2014

Replying to @mfeathers

オブジェクト指向が変化をカプセル化してコードの理解性を上げたのに対し、関数型は変化を最小化してコードの理解性を上げた。TRT "@mfeathers: OO makes code understandable by encapsulating moving..."

1 3 2

New to Twitter?

Sign up now to get your own personalized timeline!

Sign up

Relevant people

Michael Feathers @mfeathers

Director, R7K Research & Conveyance.
Author of Working Effectively with Legacy Code.

Follow

See twitter.com/mfeathers/status/29581296216?lang=en

Summary of the Programming Paradigms in Modern Java

- Summary of these two paradigms:
 - Java's object-oriented programming features make code understandable by encapsulating the moving parts



```
List<String> zap  
(List<String> lines,  
 String omit) {  
 List<String> res =  
     new ArrayList<>();  
 for (String line : lines)  
     if (!omit.equals(line))  
         res.add(line);  
 return res;  
}
```

Summary of the Programming Paradigms in Modern Java

- Summary of these two paradigms:
 - Java's object-oriented programming features make code understandable by encapsulating the moving parts
 - It's functional programming features make code understandable by eliminating the moving parts



```
List<String> zap  
(List<String> lines,  
 String omit) {  
 return lines  
 .parallelStream()  
 .filter(not(omit::equals))  
 .collect(toList());  
}
```

End of Overview of Java's Supported Programming Paradigms

Understand Java's Key Functional Programming Concepts & Features

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

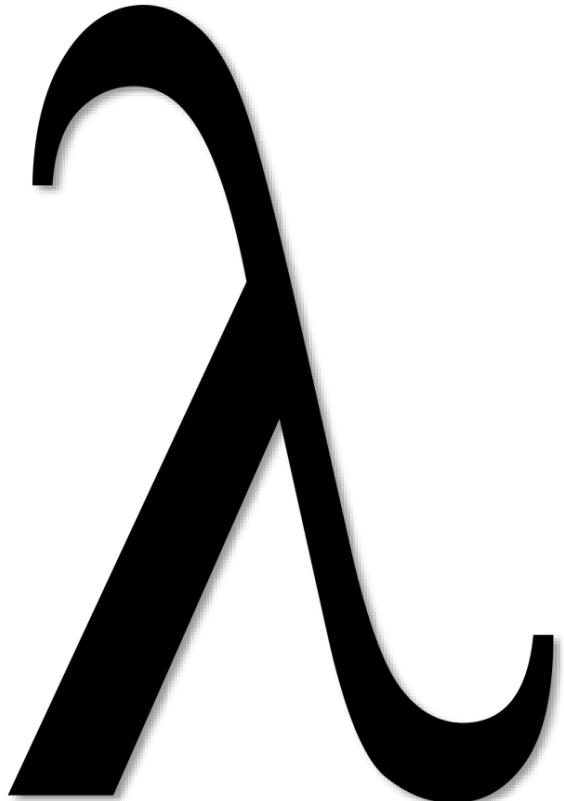
**Institute for Software
Integrated Systems**

**Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Lesson

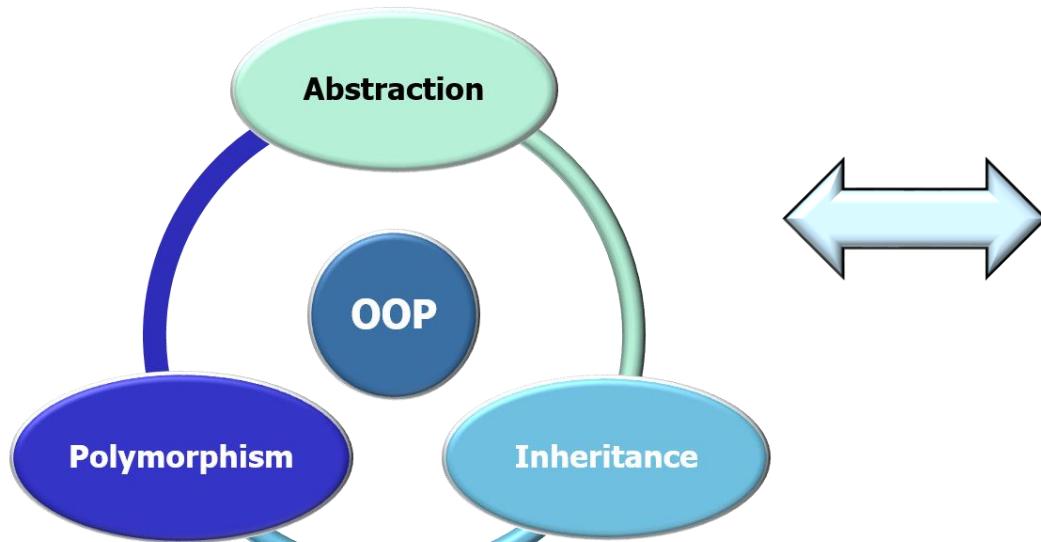
- Understand key functional programming concepts & features supported by Java



These functional programming features were added in Java 8 & expanded later

Learning Objectives in this Lesson

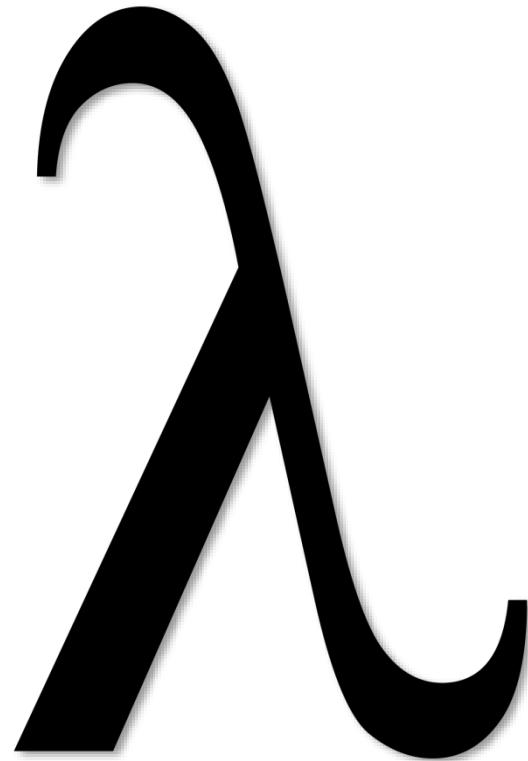
- Understand key functional programming concepts & features supported by Java
- Know how to compare & contrast functional programming & object-oriented programming



Key Functional Programming Concepts in Java

Key Functional Programming Concepts in Java

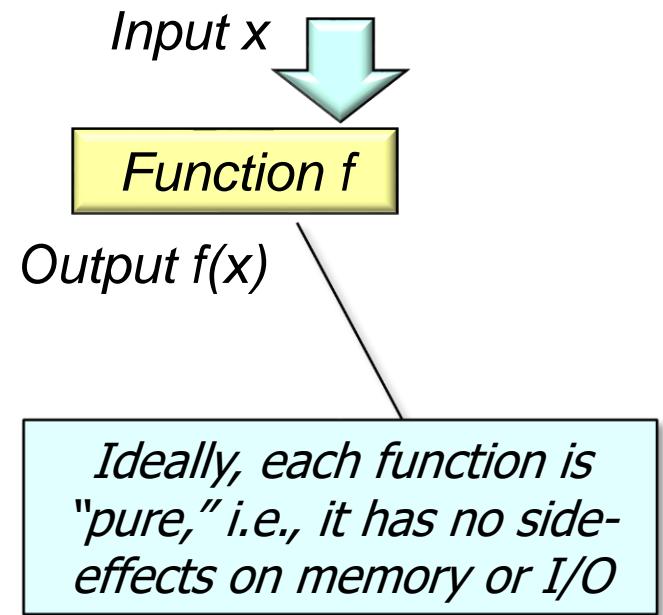
- Functional programming has its roots in lambda calculus



See en.wikipedia.org/wiki/Functional_programming

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions

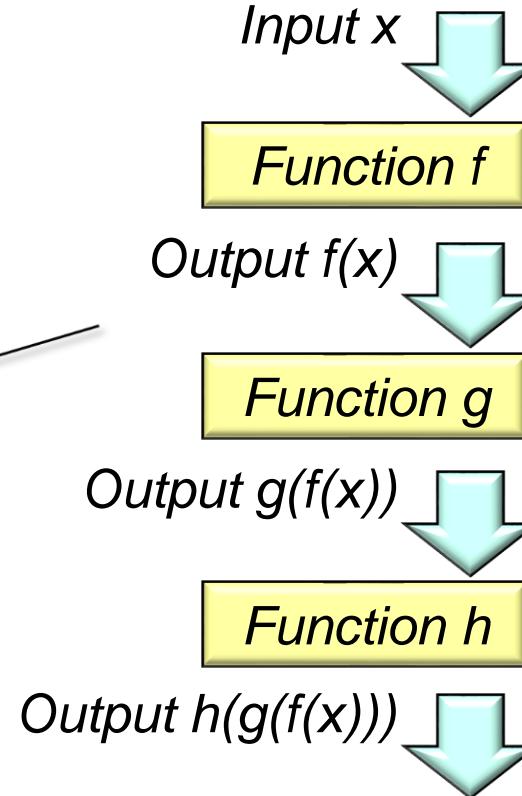


See en.wikipedia.org/wiki/Functional_programming#Pure_functions

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions

Note "function composition": the output of one function serves as the input to the next function, etc.

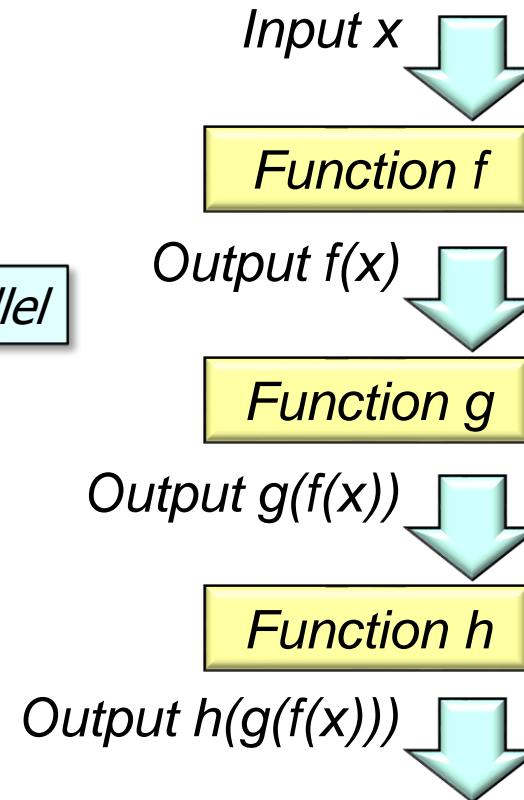


Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions

Functionally compute the 'nth' factorial in parallel

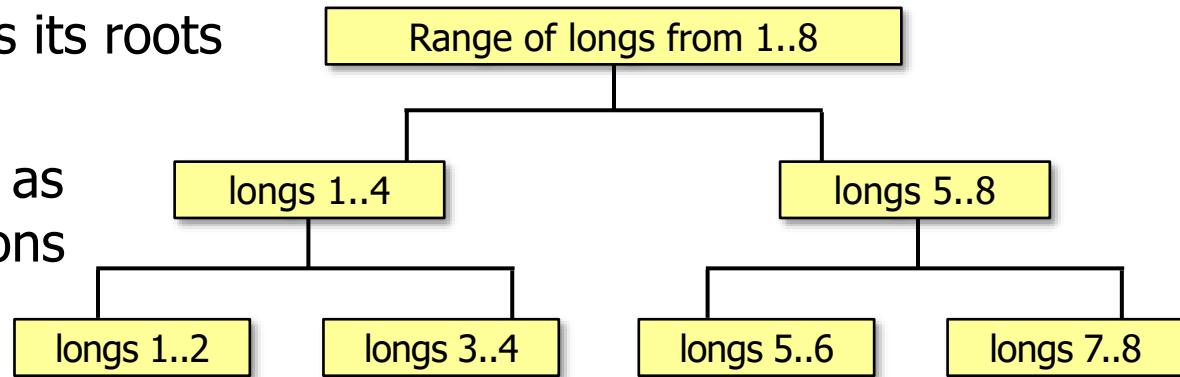
```
long factorial  
    (long n) {  
        return LongStream  
            .rangeClosed(1, n)  
            .parallel()  
            .reduce(1,  
                (a, b) -> a * b);  
    }
```



Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions

```
long factorial
    (long n) {
        return LongStream
            .rangeClosed(1, n)
            .parallel()
            .reduce(1,
                (a, b) -> a * b);
    }
```

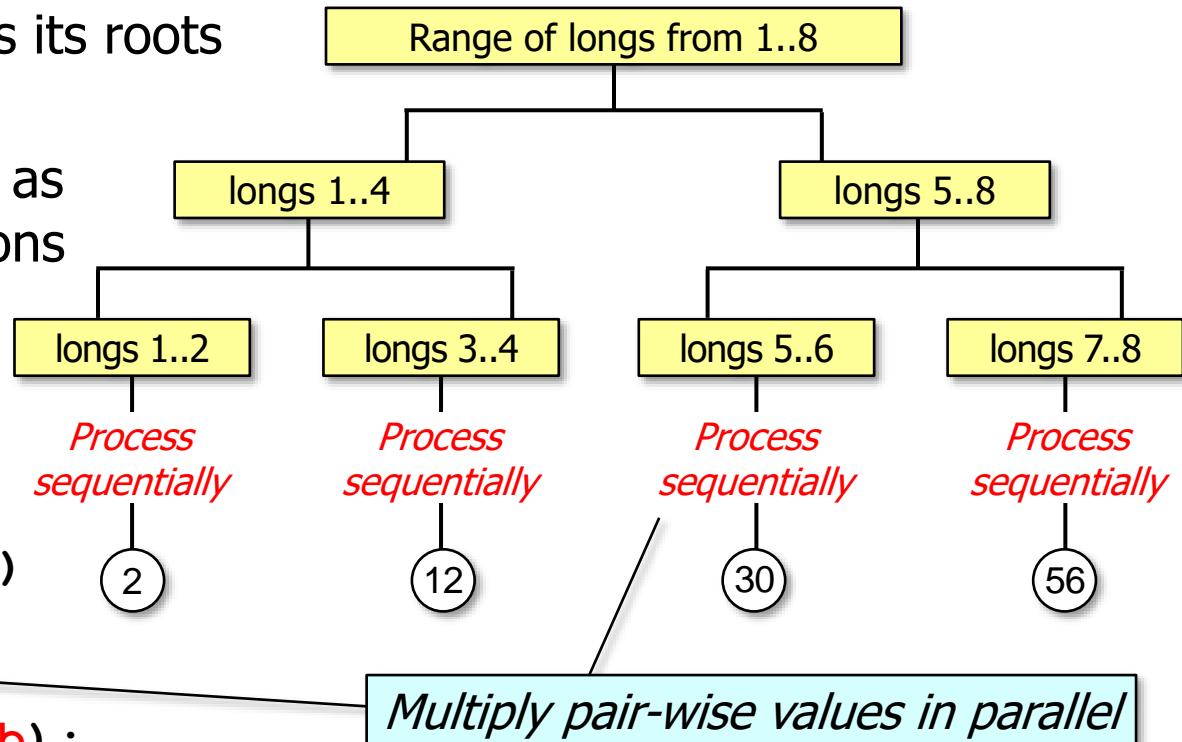


Generate a stream of longs from 1 to n in parallel (where n == 8)

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions

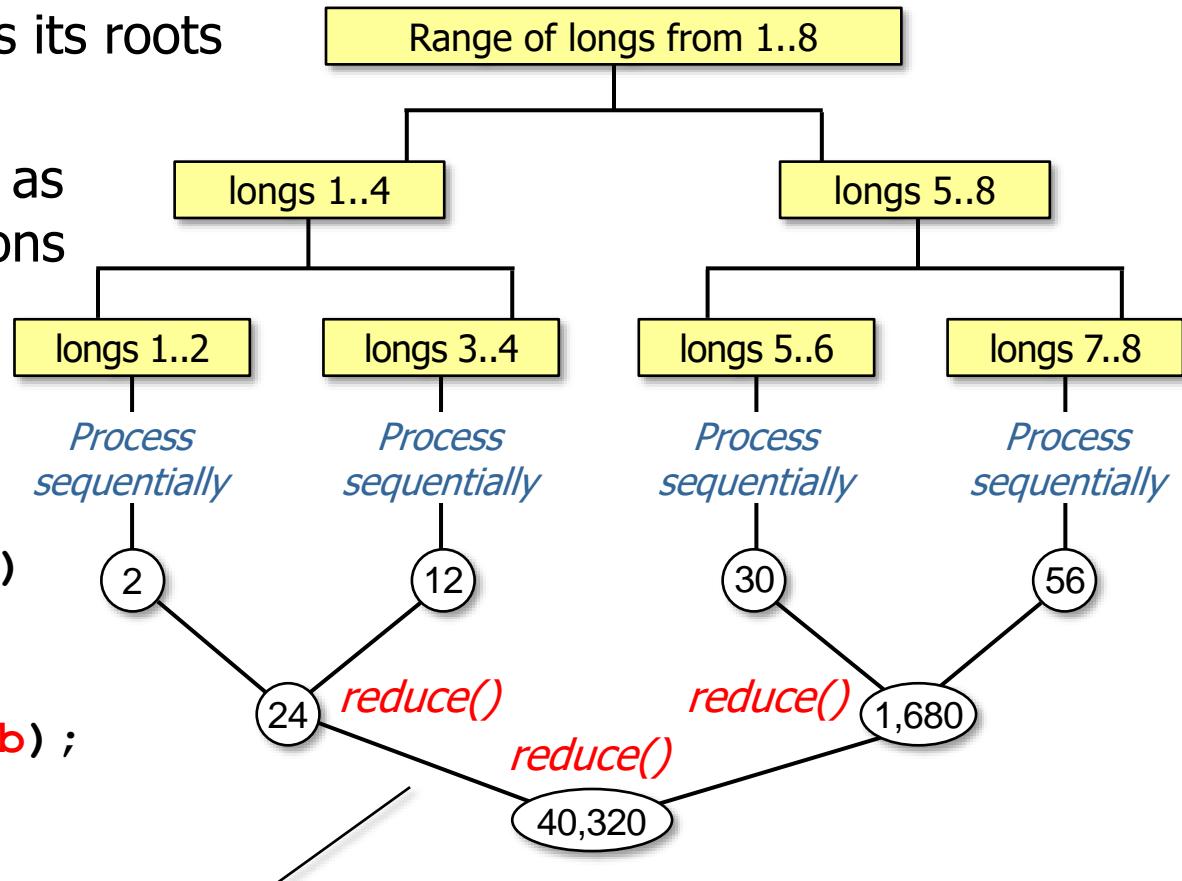
```
long factorial
    (long n) {
        return LongStream
            .rangeClosed(1, n)
            .parallel()
            .reduce(1,
                (a, b) -> a * b);
    }
```



Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions

```
long factorial
    (long n) {
        return LongStream
            .rangeClosed(1, n)
            .parallel()
            .reduce(1,
                (a, b) -> a * b);
    }
```



Successively combine two immutable long values & produce a new one

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards



See [en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,

- Computations are treated as evaluation of math functions
- Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```

Shared mutable state

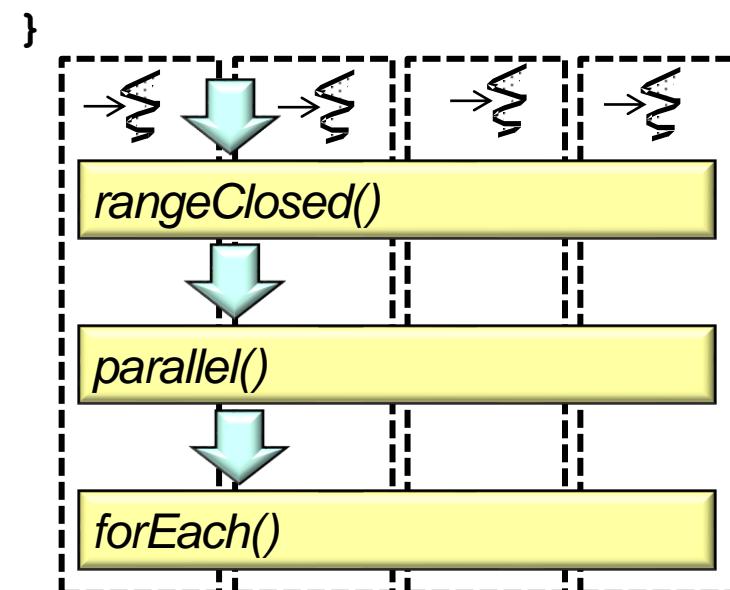


Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```



See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```

Beware of race conditions!!!



See en.wikipedia.org/wiki/Race_condition#Software

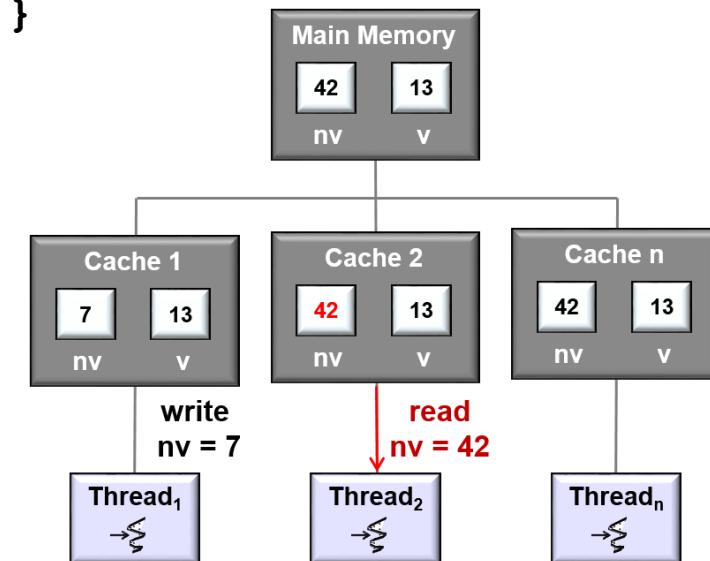
Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

Beware of inconsistent memory visibility

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```



Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards

```
long factorial(long n) {  
    Total t = new Total();  
    LongStream.rangeClosed(1, n)  
        .parallel()  
        .forEach(t::mult);  
  
    return t.mTotal;  
}
```

```
class Total {  
    public long mTotal = 1;  
  
    public void mult(long n)  
    { mTotal *= n; }  
}
```



***Only you can prevent
concurrency hazards!***

In Java *you* must avoid these hazards, i.e., the compiler & JVM won't save you..

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards
 - Instead, focus is on “immutable” objects



See docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards
 - Instead, focus is on “immutable” objects
 - Immutable object state cannot change after it is constructed

```
final class String {  
    private final char value[];  
    ...  
  
    public String(String s) {  
        value = s;  
        ...  
    }  
  
    public int length() {  
        return value.length;  
    }  
    ...  
}
```

Key Functional Programming Concepts in Java

- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards
 - Instead, focus is on “immutable” objects
 - Immutable object state cannot change after it is constructed
 - Java String is a common example of an immutable object

```
final class String {  
    private final char value[];  
    ...  
  
    public String(String s) {  
        value = s;  
        ...  
    }  
  
    public int length() {  
        return value.length;  
    }  
    ...  
}
```

Key Functional Programming Concepts in Java

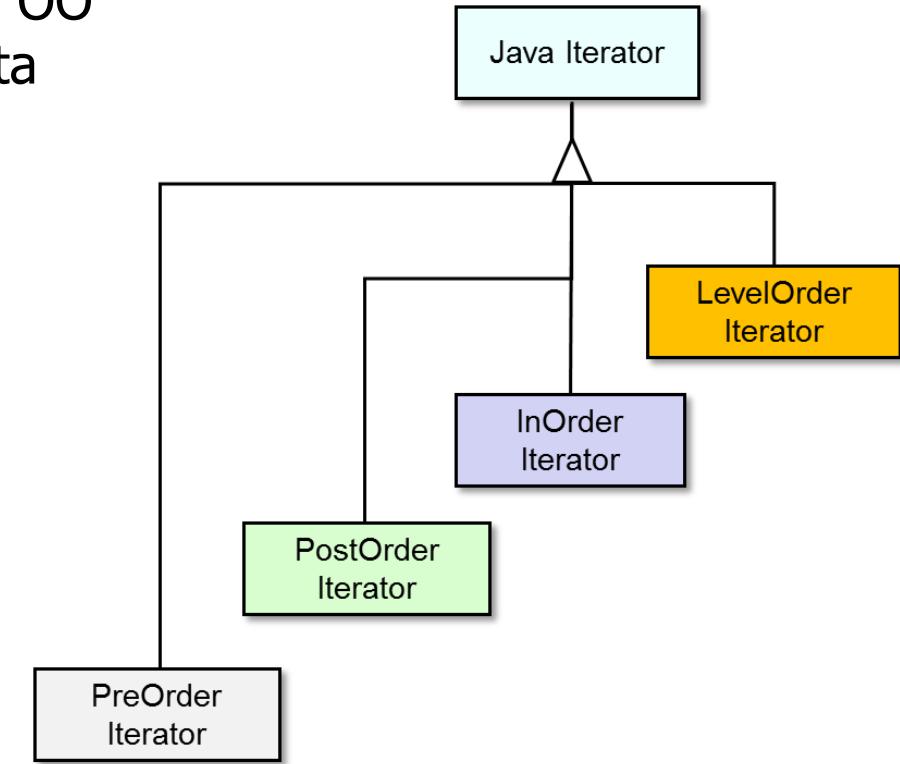
- Functional programming has its roots in lambda calculus, e.g.,
 - Computations are treated as evaluation of math functions
 - Changing state & mutable shared data are discouraged to avoid various hazards
 - Instead, focus is on “immutable” objects
 - Immutable object state cannot change after it is constructed
 - Java String is a common example of an immutable object
 - Fields are final & only accessor methods

```
final class String {  
    private final char value[];  
    ...  
  
    public String(String s) {  
        value = s;  
        ...  
    }  
  
    public int length() {  
        return value.length;  
    }  
    ...  
}
```

Functional vs. Object-Oriented Programming in Java

Functional vs. Object-Oriented Programming in Java

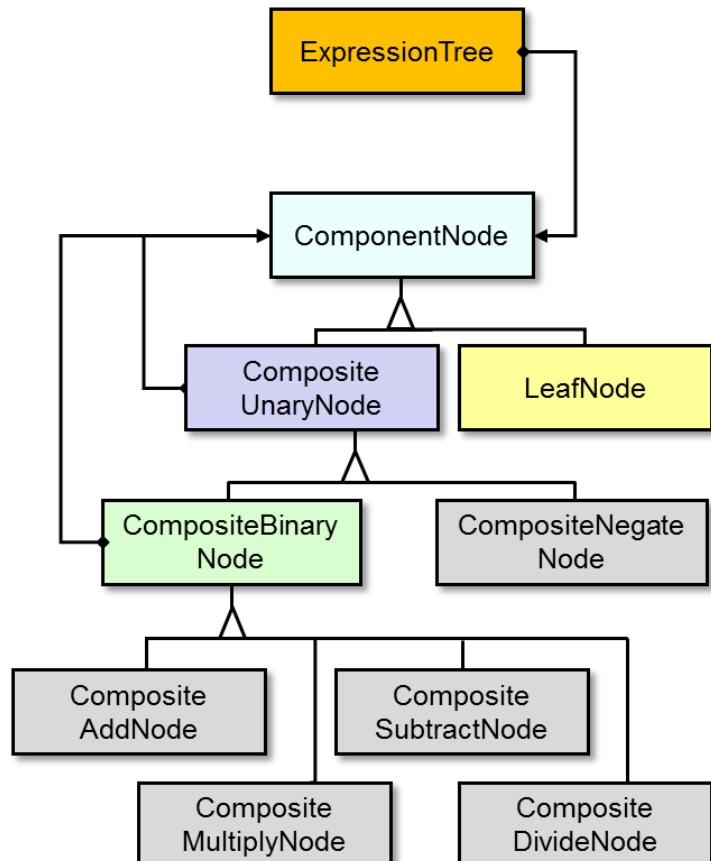
- In contrast to functional programming, OO programming employs “hierarchical data abstraction”



See en.wikipedia.org/wiki/Object-oriented_design

Functional vs. Object-Oriented Programming in Java

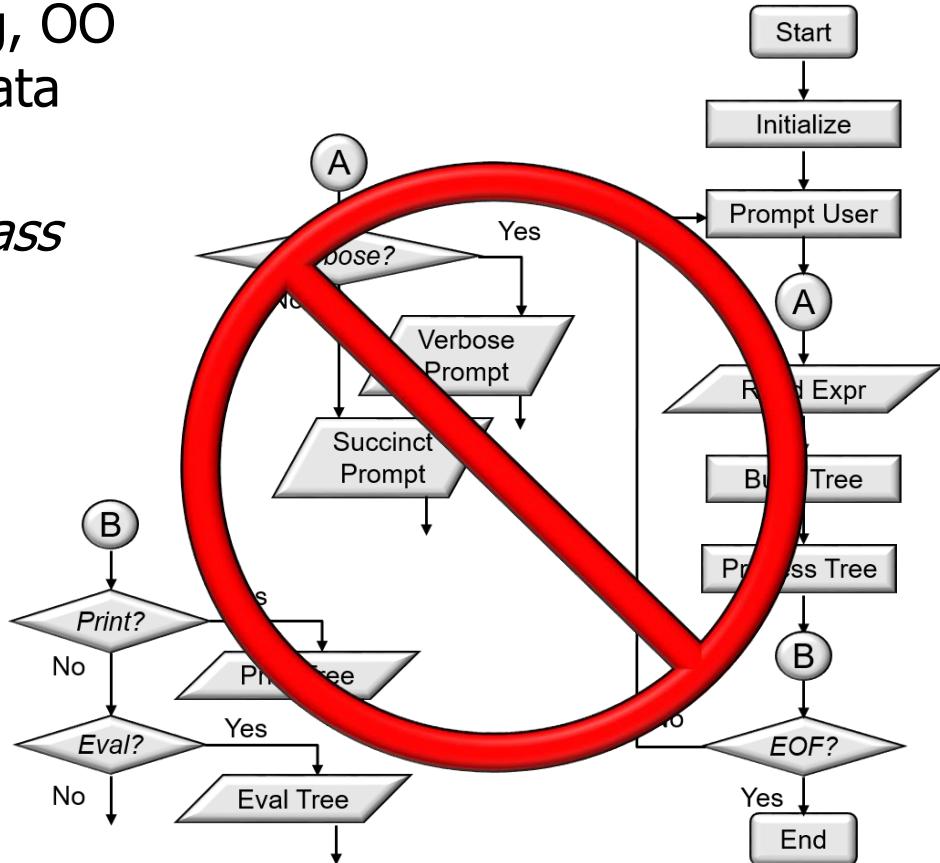
- In contrast to functional programming, OO programming employs “hierarchical data abstraction”, e.g.
 - Components are based on stable *class* roles & relationships extensible via inheritance & dynamic binding



See en.wikipedia.org/wiki/Object-oriented_programming

Functional vs. Object-Oriented Programming in Java

- In contrast to functional programming, OO programming employs “hierarchical data abstraction”, e.g.
 - Components are based on stable *class* roles & relationships extensible via inheritance & dynamic binding
 - Rather than algorithmic actions implemented as functions



Functional vs. Object-Oriented Programming in Java

- In contrast to functional programming, OO programming employs “hierarchical data abstraction”, e.g.
 - Components are based on stable *class* roles & relationships extensible via inheritance & dynamic binding
 - State is encapsulated by methods that perform imperative statements

```
Tree tree = ...;
Visitor printVisitor =
    makeVisitor(...);

for(Iterator<Tree> iter =
    tree.iterator();
    iter.hasNext();) {
    iter.next()
        .accept(printVisitor);
```

Functional vs. Object-Oriented Programming in Java

- In contrast to functional programming, OO programming employs “hierarchical data abstraction”, e.g.
 - Components are based on stable *class* roles & relationships extensible via inheritance & dynamic binding
 - State is encapsulated by methods that perform imperative statements



State is often “mutable” in OO programs

```
Tree tree = ...;  
Visitor printVisitor =  
    makeVisitor(...);  
  
for(Iterator<Tree> iter =  
    tree.iterator();  
    iter.hasNext();)  
    iter.next()  
        .accept(printVisitor);
```

*Access & update
internal iterator state*

End of Understand Java's Key Functional Programming Concepts & Features

Recognize How Java Combines Object-Oriented & Functional Programming

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Lesson

- Recognize the benefits of combining object-oriented & functional programming in Java



Again, we show modern Java code fragments we'll cover in more detail later

Learning Objectives in this Lesson

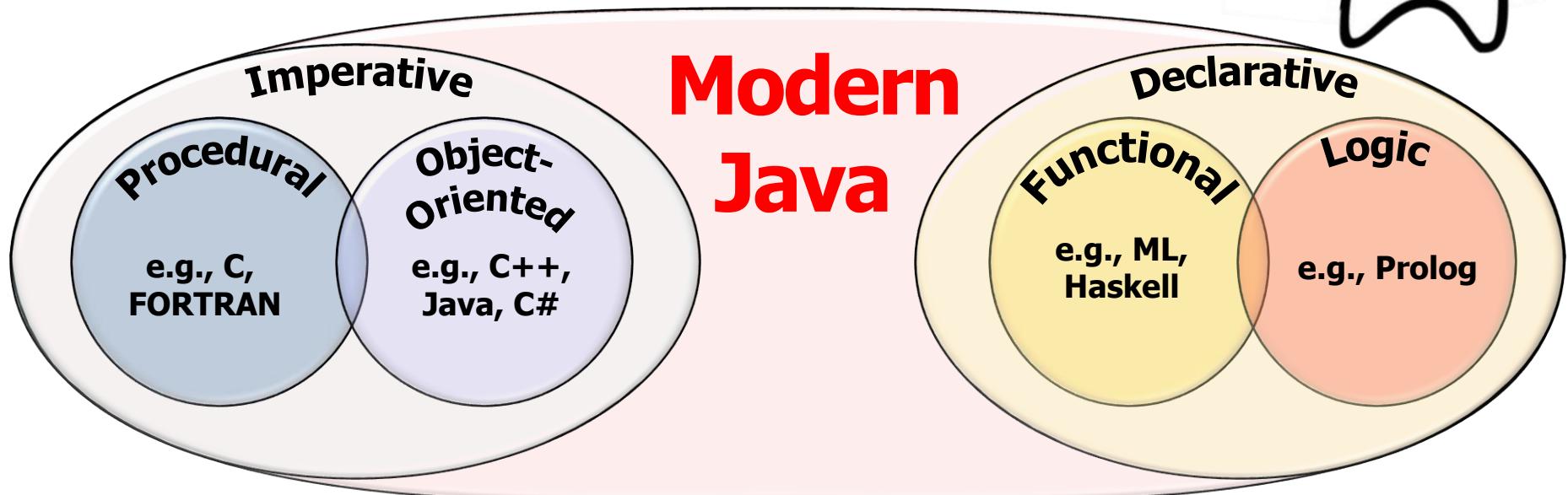
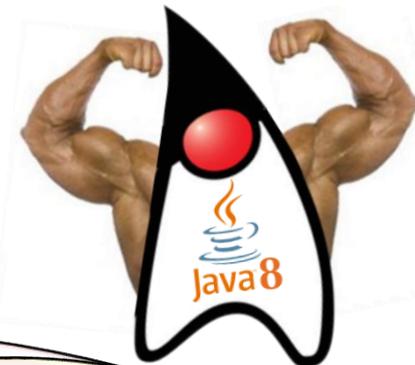
- Recognize the benefits of combining object-oriented & functional programming in Java
- Understand when, why, & how to use mutable state with Java



Combining Object-Oriented & Functional Programming in Java

Combining Object-Oriented & Functional Programming in Java

- Java's combination of functional & object-oriented paradigms is powerful!



Combining Object-Oriented & Functional Programming in Java

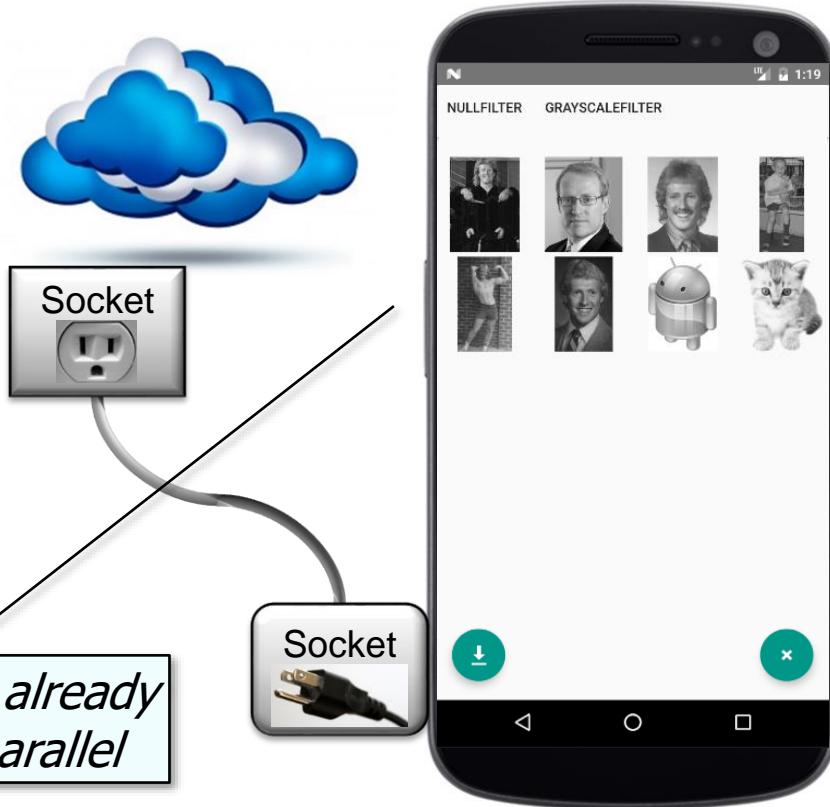
- Java's functional features help close the gap between a program's "domain intent" & its computations



See www.toptal.com/software/declarative-programming

Combining Object-Oriented & Functional Programming in Java

- Java's functional features help close the gap between a program's "domain intent" & its computations, e.g.,
 - Domain intent defines "what"

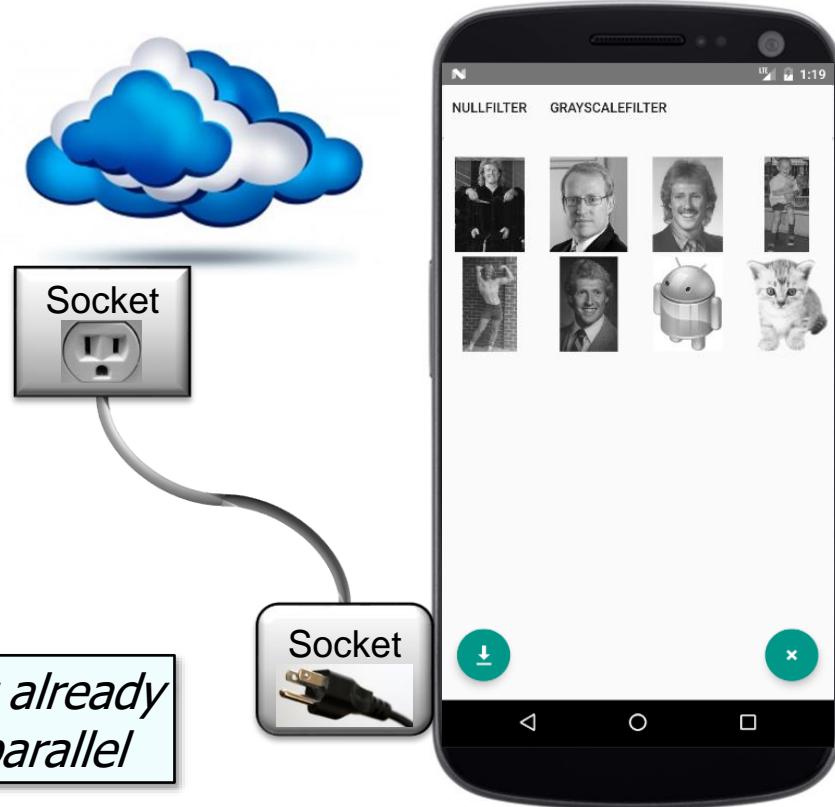


Combining Object-Oriented & Functional Programming in Java

- Java's functional features help close the gap between a program's "domain intent" & its computations, e.g.,
 - Domain intent defines "what"
 - Computations define "how"

```
List<Image> images = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```

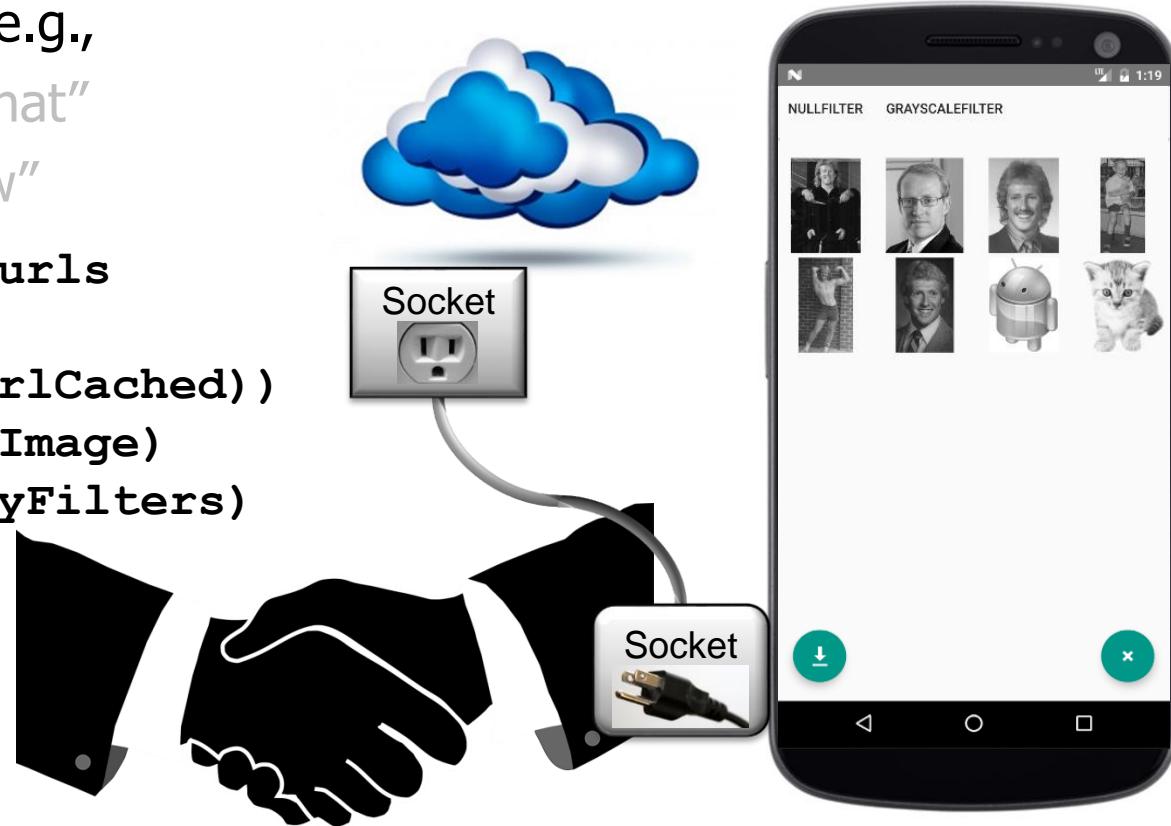
Process a list of URLs to images that aren't already cached & transform/store the images in parallel



Combining Object-Oriented & Functional Programming in Java

- Java's functional features help close the gap between a program's "domain intent" & its computations, e.g.,
 - Domain intent defines "what"
 - Computations define "how"

```
List<Image> images = urls
    .parallelStream()
    .filter(not(this::urlCached))
    .map(this::downloadImage)
    .flatMap(this::applyFilters)
    .collect(toList());
```

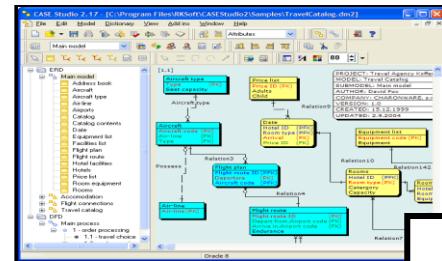


Java functional programming features connect domain intent & computations

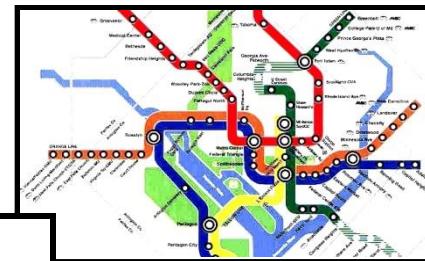
Combining Object-Oriented & Functional Programming in Java

- Likewise, Java's object-oriented features help to structure a program's software architecture

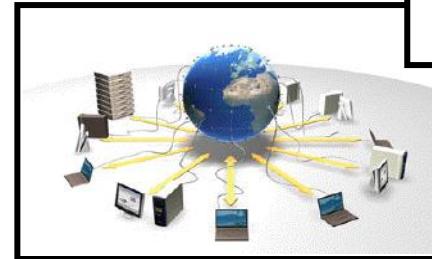
Logical View



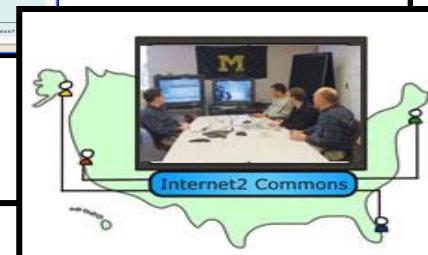
Process View



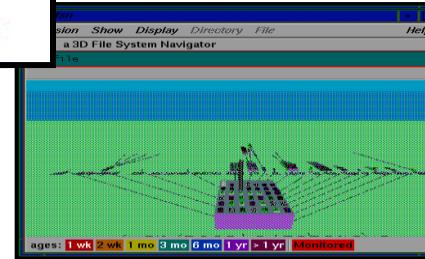
Physical View



Use Case View



Development View

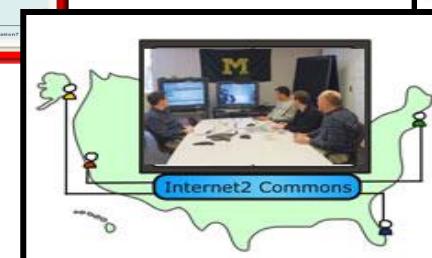
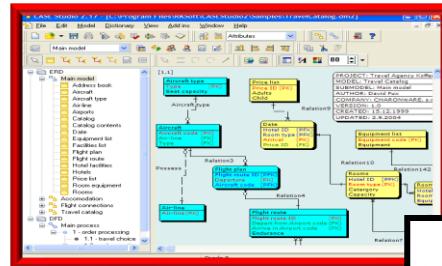


See en.wikipedia.org/wiki/Software_architecture

Combining Object-Oriented & Functional Programming in Java

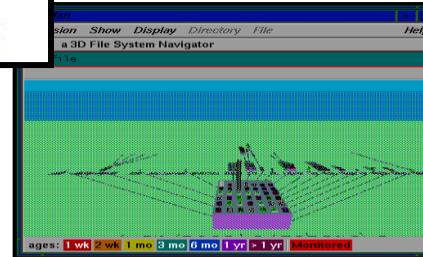
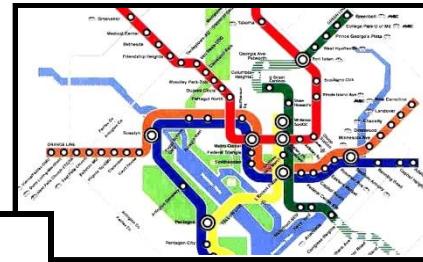
- Likewise, Java's object-oriented features help to structure a program's software architecture

Logical View



This view depicts sub-systems, packages, & classes that exhibit architecturally important structure & behavior

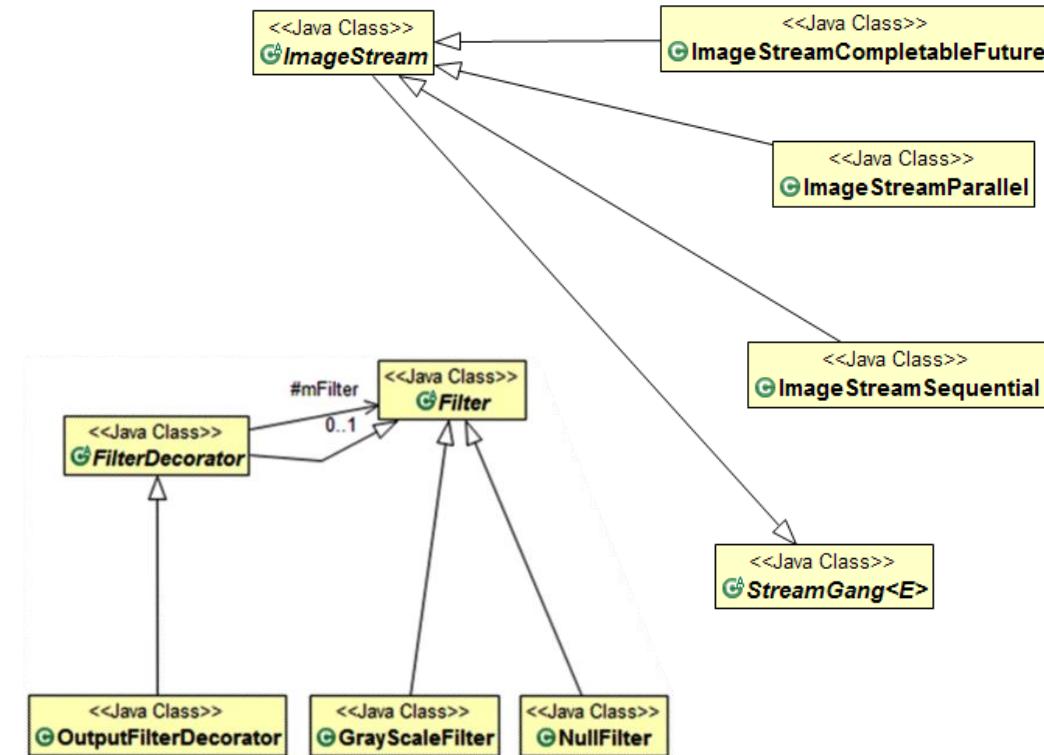
Process View



Development View

Combining Object-Oriented & Functional Programming in Java

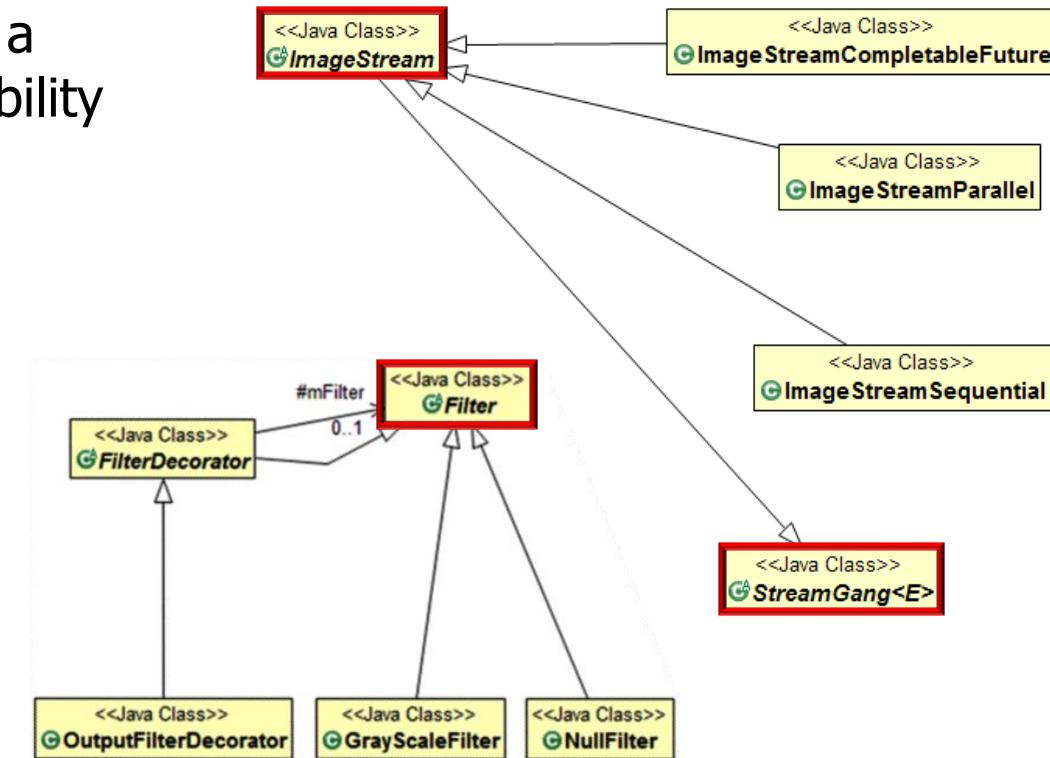
- e.g., consider the ImageStreamGang program



See github.com/douglasraigschmidt/LiveLessons/tree/master/ImageStreamGang

Combining Object-Oriented & Functional Programming in Java

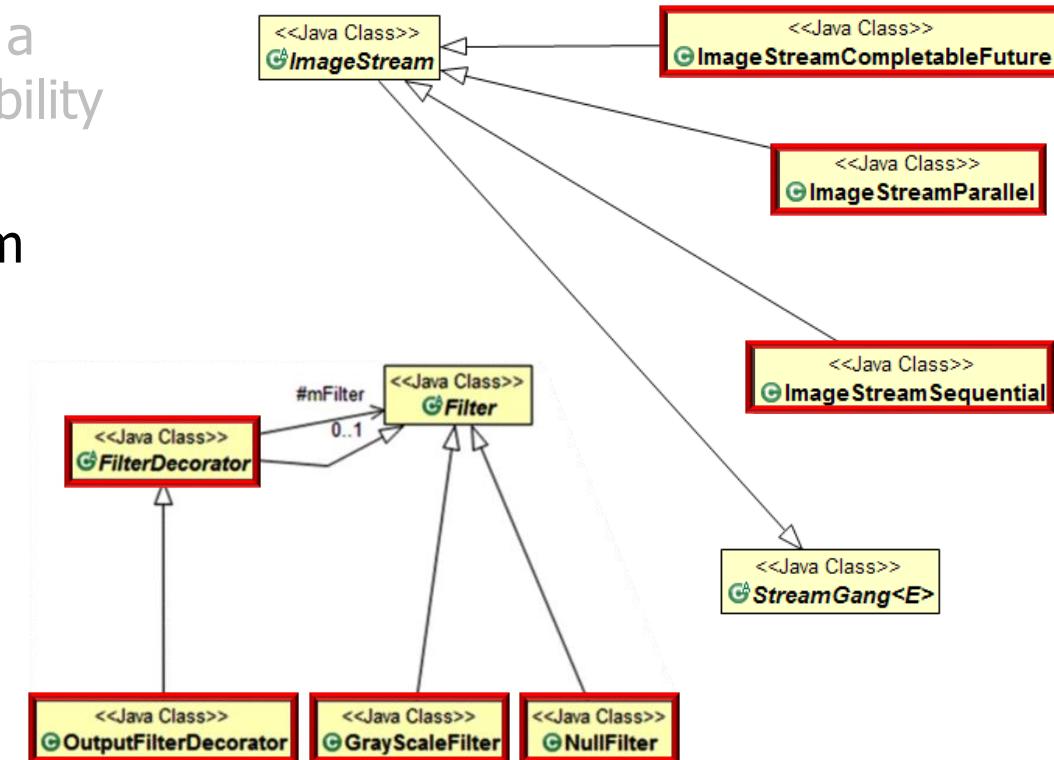
- e.g., consider the ImageStreamGang program
 - Common super classes provide a reusable foundation for extensibility



Combining Object-Oriented & Functional Programming in Java

- e.g., consider the ImageStreamGang program

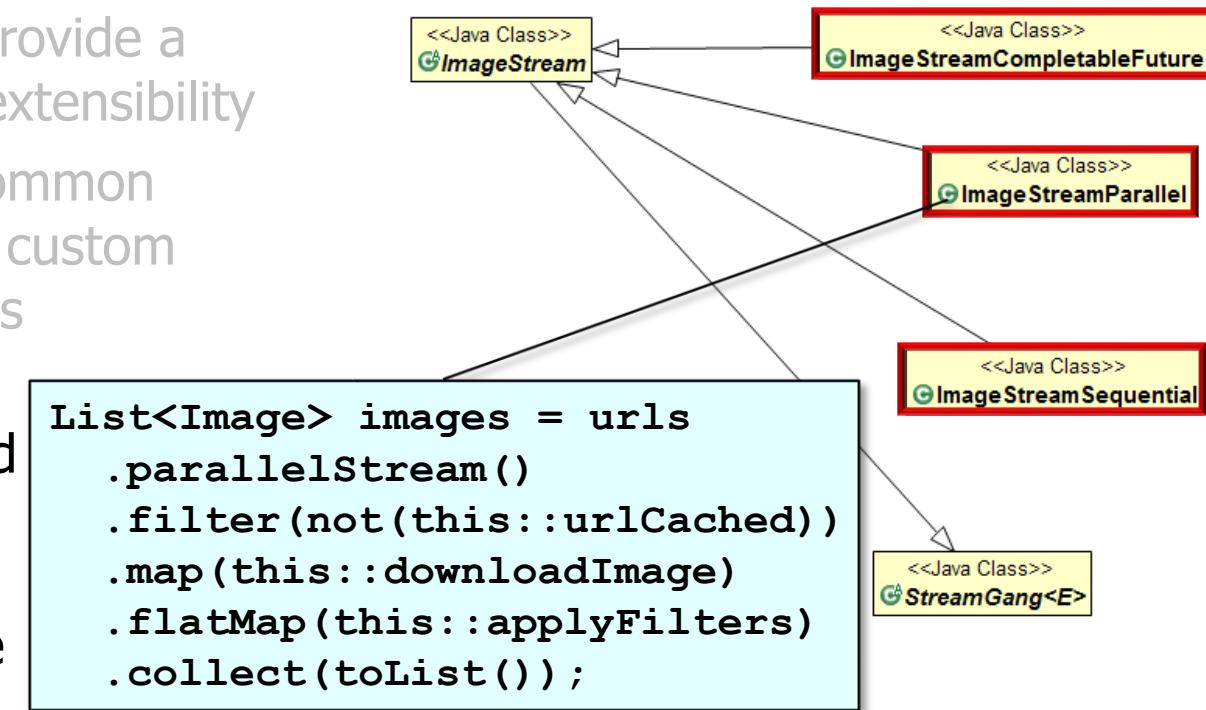
- Common super classes provide a reusable foundation for extensibility
- Subclasses extend the common classes to create various custom implementation strategies



Combining Object-Oriented & Functional Programming in Java

- e.g., consider the ImageStreamGang program

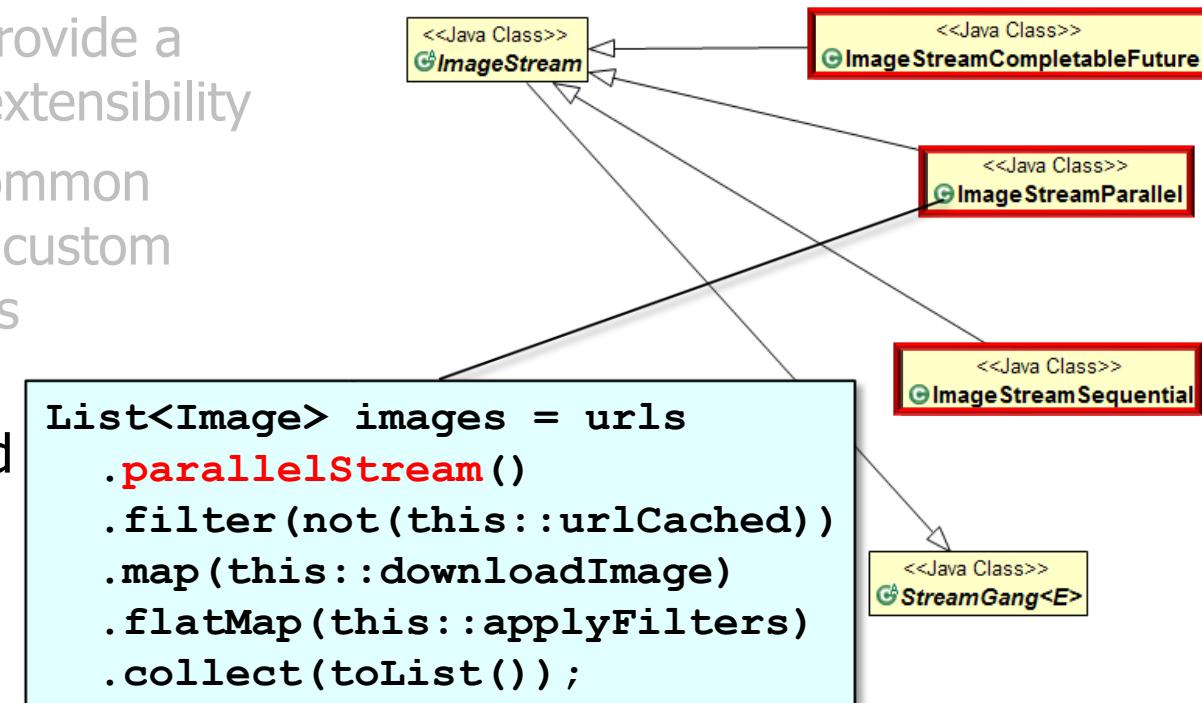
- Common super classes provide a reusable foundation for extensibility
- Subclasses extend the common classes to create various custom implementation strategies
- Java's FP features are most effective when used to simplify computations within the context of an OO software architecture



Combining Object-Oriented & Functional Programming in Java

- e.g., consider the ImageStreamGang program

- Common super classes provide a reusable foundation for extensibility
- Subclasses extend the common classes to create various custom implementation strategies
- Java's FP features are most effective when used to simplify computations within the context of an OO software architecture
 - Especially concurrent & parallel computations



See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

When, Why, & How to Use Mutable State in Java

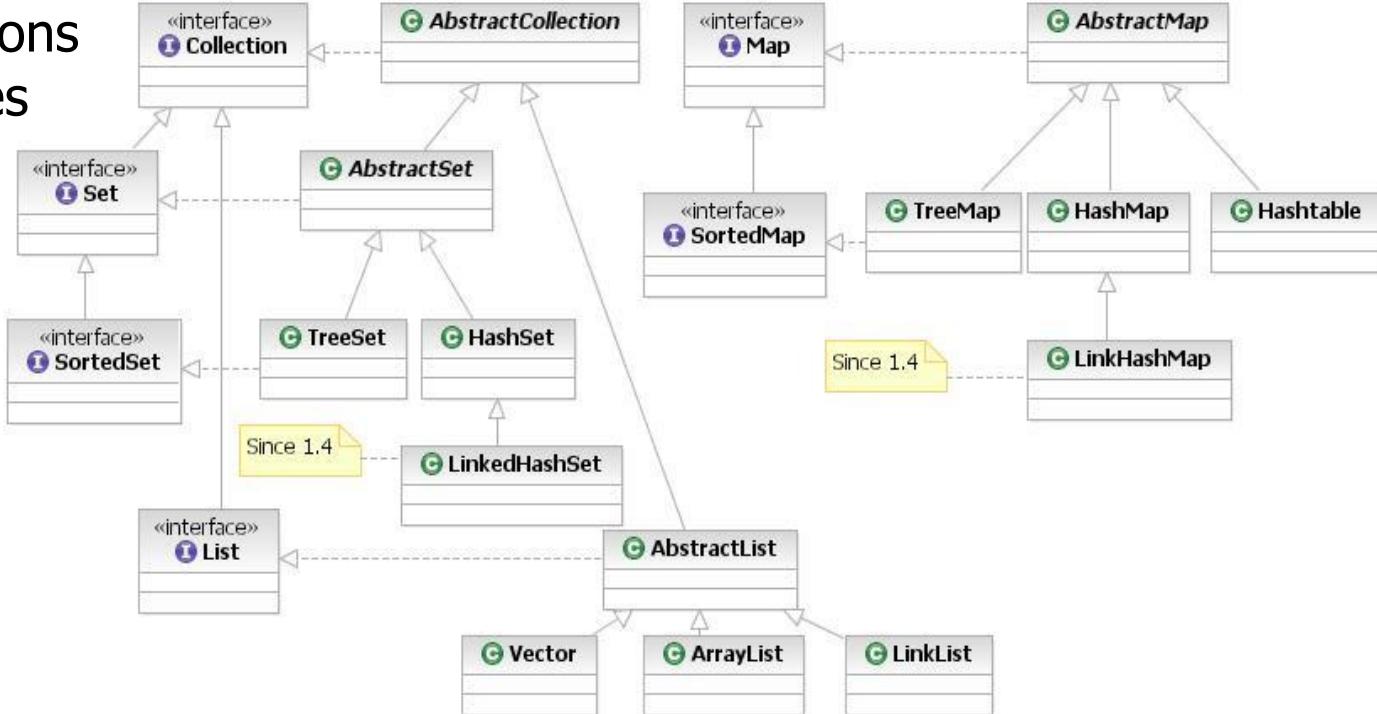
When, Why, & How to Use Mutable State in Java

- Since Java is a hybrid language, there are situations in which mutable changes to shared state are allowed/encouraged



When, Why, & How to Use Mutable State in Java

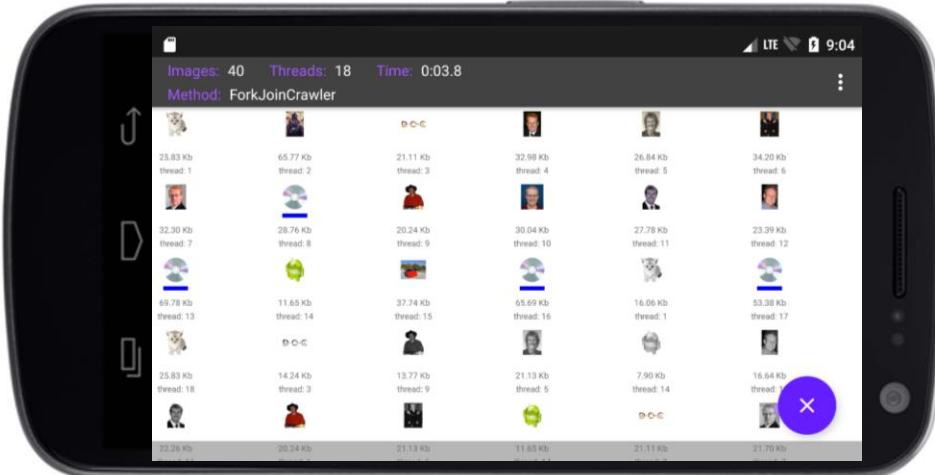
- Since Java is a hybrid language, there are situations in which mutable changes to shared state are allowed/encouraged
 - e.g., Java collections framework classes



See docs.oracle.com/javase/8/docs/technotes/guides/collections/

When, Why, & How to Use Mutable State in Java

- However, you're usually better off by minimizing/avoiding the use of shared mutable state in *your* programs!!



When, Why, & How to Use Mutable State in Java

- If you *do* share mutable state in your programs then make sure you add the necessary synchronizers and/or use concurrent/synchronized collections

Category	Definition	
Atomic operations	An action that effectively happens all at once or not at all	
Mutual exclusion	Allows concurrent access & updates to shared mutable data without race conditions	
Coordination	Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.	
Barrier synchronization	Ensures that any thread(s) must stop at a certain point & cannot proceed until all other thread(s) reach this barrier	

See www.youtube.com/playlist?list=PLZ9NgFYEMxp6IM0Cddzr_qjqfiGC2pq1a

End of Recognize How Java Combines Object-Oriented & Functional Programming

Overview of Java Lambda Expressions

Douglas C. Schmidt

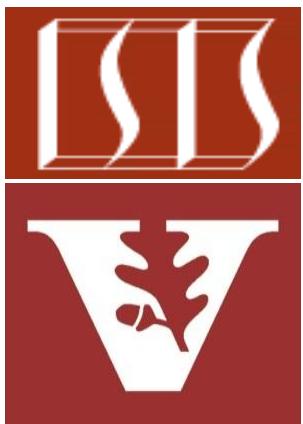
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Lesson

- Understand foundational functional programming features in Java, e.g.,
 - Lambda expressions



Several examples showcase foundational Java functional programming features

Overview of Java Lambda Expressions

Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later

```
new Thread(() ->  
    System.out.println("hello world"))  
.start();
```

Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later

```
new Thread( () ->  
    System.out.println("hello world") )  
.start();
```

The Thread constructor expects an instance of Runnable.

Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread( () ->  
    System.out.println("hello world"))  
    .start();
```

This lambda expression takes no parameters, i.e., "()"

Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread(() -> ————— Arrow separates the param list from the lambda body.  
    System.out.println("hello world"))  
    .start();
```

Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread(() ->  
    System.out.println("hello world"))  
.start();
```

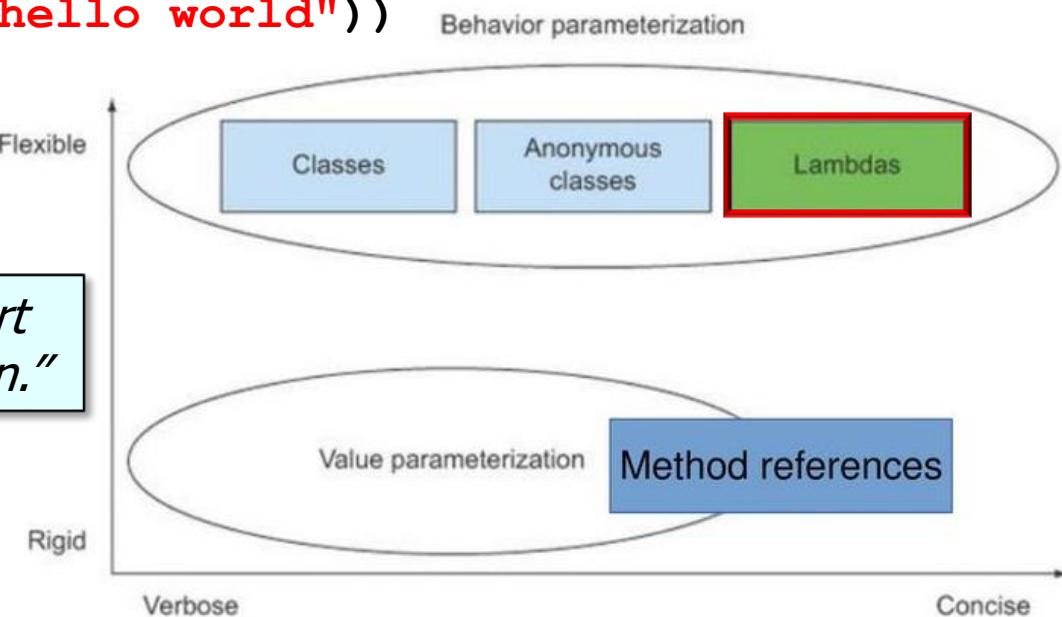
The lambda body defines the computation.

Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread(() ->  
    System.out.println("hello world"))  
.start();
```

Java's lambda expressions support concise "behavior parameterization."



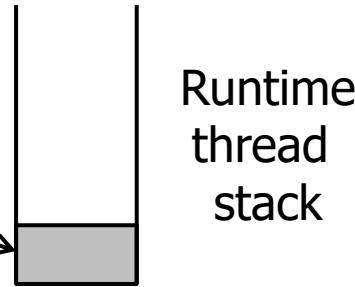
See blog.indrek.io/articles/java-8-behavior-parameterization

Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread(() ->  
    System.out.println("hello world"))  
.start();
```

*This lambda defines a computation
that runs in a separate Java thread.*



Runtime
thread
stack

Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread( () ->  
    System.out.println("hello world"))  
    .start();
```

```
Runnable r = () -> System.out.println("hello world");  
new Thread(r).start();
```

You can also store a lambda expression into
a variable & pass that variable to a method

Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread() ->  
    System.out.println("hello world"))  
    .start();
```

Lambda expressions are compact since they just focus on computation(s) to perform.



Overview of Java Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```
new Thread( () ->  
    System.out.println("hello world"))  
.start();
```

vs

Conversely, this anonymous inner class requires more code to write each time

```
new Thread(new Runnable() {  
    public void run() {  
        System.out.println("hello world");  
    } }).start();
```



Overview of Java Lambda Expressions

- A lambda expression can access (effectively) final variables from the enclosing scope

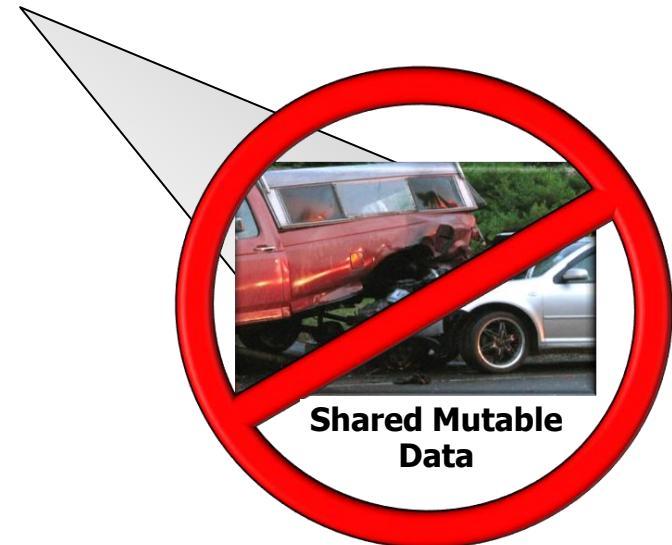
```
int answer = 42;  
new Thread(() ->  
    System.out.println("The answer is " + answer))  
.start();
```

This lambda expression can access the value of "answer," which is an effectively final variable whose value never changes after it's initialized

Overview of Java Lambda Expressions

- Lambda expressions are most effective when they are “stateless” & have no shared mutable data.

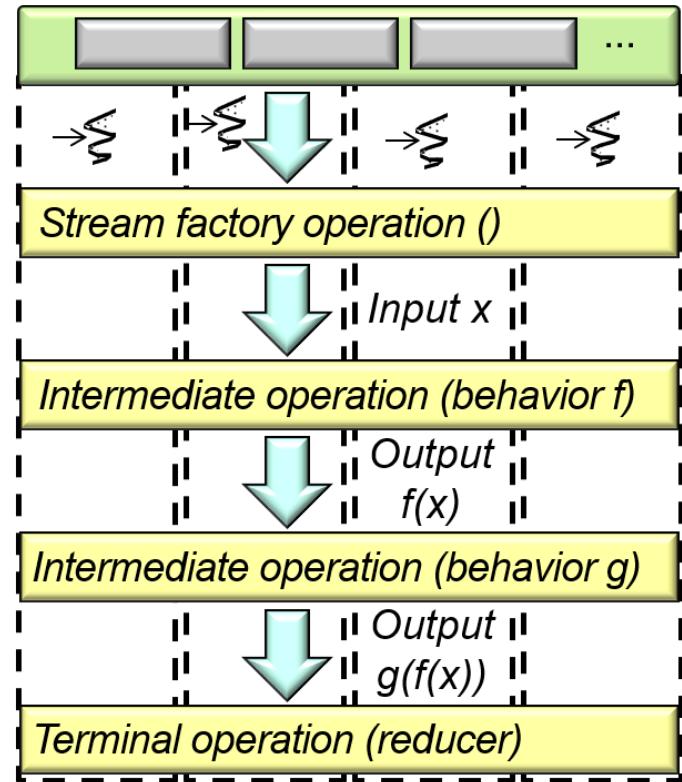
```
int answer = 42;  
new Thread(() -> System.out.println("The answer is " + answer))  
.start();
```



Overview of Java Lambda Expressions

- Lambda expressions are most effective when they are “stateless” & have no shared mutable data.

*Stateless lambda expressions
are particularly useful when
applied to Java parallel streams.*



Benefits of Lambda Expressions

Benefits of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```

Benefits of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes, e.g.

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

Array of names represented as strings

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
    (s, t) -> s.compareToIgnoreCase(t));
```

Benefits of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes, e.g.

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```



*Extraneous syntax for
anonymous inner class*

Benefits of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes, e.g.

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```



(s, t) is short for *(String s, String t)*, which leverages Java's type inference capabilities.

See docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html

Benefits of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes, e.g.

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
    (s, t) -> s.compareToIgnoreCase(t));
```



This lambda expression omits the method name & extraneous syntax.

Benefits of Lambda Expressions

- Lambda expressions can work with multiple parameters in a *much* more compact manner than anonymous inner classes, e.g.

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
    (s, t) -> s.compareToIgnoreCase(t));
```



Therefore, it's good practice to use lambda expressions whenever you can!

End of Overview of Java Lambda Expressions

Overview of Java Method References

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Lesson

- Understand foundational functional programming features in Java, e.g.,
 - Lambda expressions
 - Method (& constructor) references



Several examples showcase foundational Java function programming features

Overview of Method References

Overview of Method References

- A compact, easy-to-read handle for a method that already has a name

Kind	Syntax	Method Reference	Lambda Expression
1. Reference to a static method	ContainingClass::staticMethodName	String::valueOf	s -> String.valueOf(s)
2. Reference to an instance method of a particular object	containingObject::instanceMethodName	s::toString	s -> s.toString()
3. Reference to instance method of an arbitrary object of a given type	ContainingType::methodName	String::toString	s -> s.toString()
4. Reference to a constructor	ClassName::new	String::new	() -> new String()

See docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

Overview of Method References

- A compact, easy-to-read handle for a method that already has a name
 - It's shorthand syntax for a lambda expression that executes one method

Kind	Syntax	Method Reference	Lambda Expression
1. Reference to a static method	ContainingClass:: staticMethodName	String::valueOf	s -> String.valueOf(s)
2. Reference to an instance method of a particular object	containingObject:: instanceMethodName	s::toString	s -> s.toString()
3. Reference to instance method of an arbitrary object of a given type	ContainingType:: methodName	String::toString	s -> s.toString()
4. Reference to a constructor	ClassName::new	String::new	() -> new String()

Overview of Method References

- A compact, easy-to-read handle for a method that already has a name
 - It's shorthand syntax for a lambda expression that executes one method

Kind	Syntax	Method Reference	Lambda Expression
1. Reference to a static method	ContainingClass:: staticMethodName	String::valueOf	s -> String.valueOf(s)
2. Reference to an instance method of a particular object	containingObject:: instanceMethodName	s::toString	s -> s.toString()
3. Reference to instance method of an arbitrary object of a given type	ContainingType:: methodName	String::toString	s -> s.toString()
4. Reference to a constructor	ClassName::new	String::new	() -> new String()

Overview of Method References

- A compact, easy-to-read handle for a method that already has a name
 - It's shorthand syntax for a lambda expression that executes one method

Kind	Syntax	Method Reference	Lambda Expression
1. Reference to a static method	ContainingClass::staticMethodName	String::valueOf	s -> String.valueOf(s)
2. Reference to an instance method of a particular object	containingObject::instanceMethodName	s::toString	s -> s.toString()
3. Reference to instance method of an arbitrary object of a given type	ContainingType::methodName	String::toString	s -> s.toString()
4. Reference to a constructor	ClassName::new	String::new	() -> new String()

Overview of Method References

- A compact, easy-to-read handle for a method that already has a name
 - It's shorthand syntax for a lambda expression that executes one method

Kind	Syntax	Method Reference	Lambda Expression
1. Reference to a static method	ContainingClass:: staticMethodName	String::valueOf	s -> String.valueOf(s)
2. Reference to an instance method of a particular object	containingObject:: instanceMethodName	s::toString	s -> s.toString()
3. Reference to instance method of an arbitrary object of a given type	ContainingType:: methodName	String::toString	s -> s.toString()
4. Reference to a constructor	ClassName::new	String::new	() -> new String()

Overview of Method References

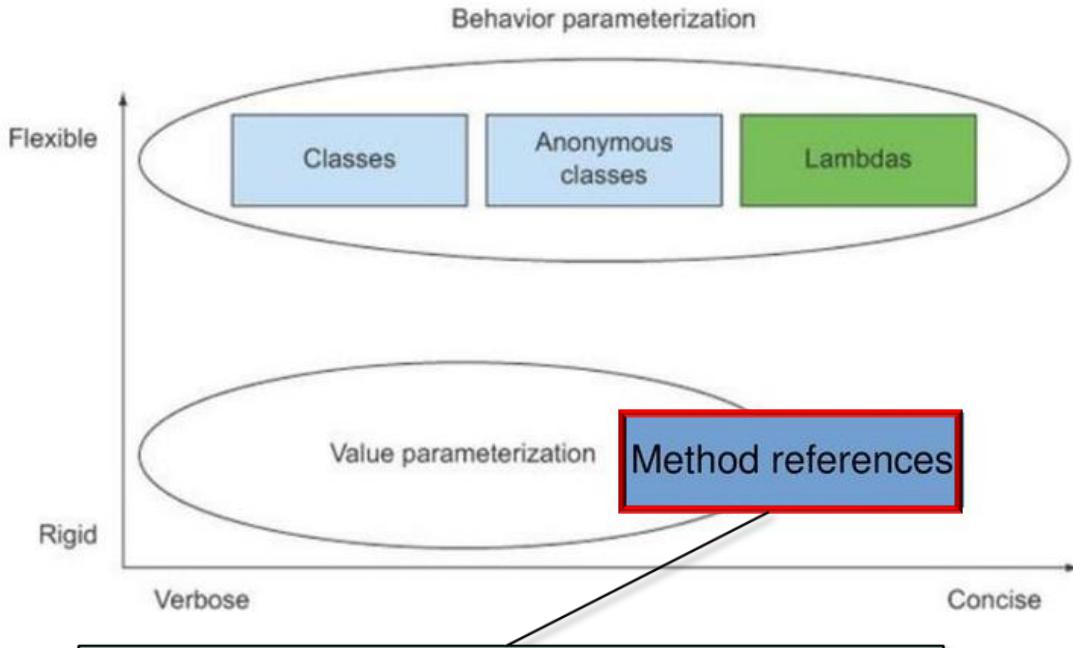
- A compact, easy-to-read handle for a method that already has a name
 - It's shorthand syntax for a lambda expression that executes one method

Kind	Syntax	Method Reference	Lambda Expression
1. Reference to a static method	ContainingClass:: staticMethodName	String::valueOf	s -> String.valueOf(s)
2. Reference to an instance method of a particular object	containingObject:: instanceMethodName	s::toString	s -> s.toString()
3. Reference to instance method of an arbitrary object of a given type	ContainingType:: methodName	String::toString	s -> s.toString()
4. Reference to a constructor	ClassName::new	String::new	() -> new String()

Benefits of Method References

Benefits of Method References

- Method references are more compact than alternative mechanisms



Java method references support more concise "behavior parameterization"

See blog.indrek.io/articles/java-8-behavior-parameterization

Benefits of Method References

- Method references are more compact than alternative mechanisms, e.g.,

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```

VS

Method references are even more compact & readable

```
Arrays.sort(nameArray, String::compareToIgnoreCase);
```



Benefits of Method References

- Method references are more compact than alternative mechanisms, e.g.,

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```

VS

Method references also promote code reuse

```
Arrays.sort(nameArray, String::compareToIgnoreCase);
```



The Arrays.sort() implementation doesn't change, but the params do!

Benefits of Method References

- Method references are more compact than alternative mechanisms, e.g.,

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```



vs *Replacing one comparison with another is easy, a la the Strategy pattern.*



```
Arrays.sort(nameArray, String::compareTo);
```

See en.wikipedia.org/wiki/Strategy_pattern

Benefits of Method References

- Method references are more compact than alternative mechanisms, e.g.,

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

```
Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return
        s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```
Arrays.sort(nameArray,
            (s, t) -> s.compareToIgnoreCase(t));
```



VS

```
Arrays.sort(nameArray, String::compareTo);
```



It's therefore good practice to use method references whenever you can!

Applying Method References in Practice

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

Array of names represented as strings

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array

```
System.out.println(List.of(nameArray));
```

prints

```
[Barbara, James, Mary, John, Linda, Michael, Linda, james, mary]
```

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array

```
System.out.println(List.of(nameArray));
```

prints

Factory method returns a fixed-size list backed by the array.

```
[Barbara, James, Mary, John, Linda, Michael, Linda, james, mary]
```

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java's forEach() methods can be used to print out values of an array

See www.javaworld.com/article/2461744/java-language/java-language-iterating-over-collections-in-java-8.html

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java's forEach() methods can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference

```
Stream.of(nameArray).forEach(System.out::print);
```

prints

Factory method that creates a stream from an array

BarbaraJamesMaryJohnLindaMichaelLindajamesmary

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java's forEach() methods can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference

```
Stream.of(nameArray).forEach(System.out::print);
```

prints

BarbaraJamesMaryJohnLindaMichaelLindajamesmary

Performs method reference action on each stream element

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java's forEach() methods can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference
 - In conjunction with a collection (e.g., List)

```
List.of(nameArray).forEach(System.out::print);
```

prints

Factory method converts an array into a list.

BarbaraJamesMaryJohnLindaMichaelLinda james mary

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java's forEach() methods can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference
 - In conjunction with a collection (e.g., List)

```
List.of(nameArray).forEach(System.out::print);
```

prints

Performs method reference action on each list element

BarbaraJamesMaryJohnLindaMichaelLinda james mary

Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java's forEach() methods can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference
 - In conjunction with a collection (e.g., List)
 - forEach() on a stream differs slightly from
forEach() on a collection



Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
    "Robert", "Michael", "Linda", "james", "mary"};
```

- System.out.println() can be used to print out an array
- Java's forEach() methods can be used to print out values of an array, e.g.
 - In conjunction with a stream & method reference
 - In conjunction with a collection (e.g., List)
 - forEach() on a stream differs slightly from forEach() on a collection
 - e.g., forEach() ordering is undefined on a stream, whereas it's defined for a collection



End of Overview of Java Method References

Understand Java Functional Interfaces: Overview

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

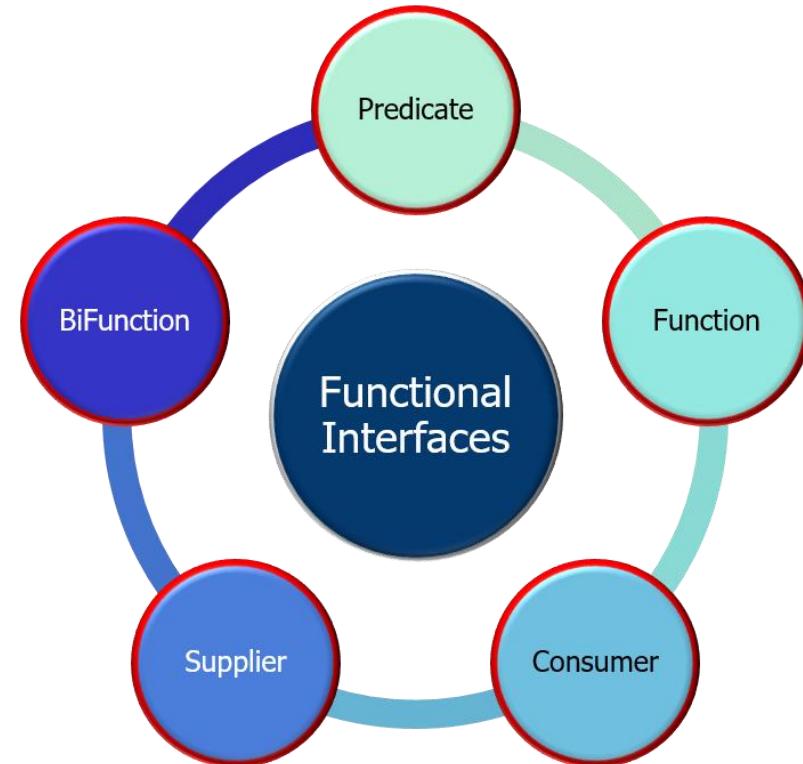
**Institute for Software
Integrated Systems**

**Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Part of the Lesson

- Understand foundational functional programming features in Java 8, e.g.,
 - Lambda expressions
 - Method & constructor references
 - Key functional interfaces



Learning Objectives in this Part of the Lesson

- Understand foundational functional programming features in Java 8, e.g.,
 - Lambda expressions
 - Method & constructor references
 - Key functional interfaces

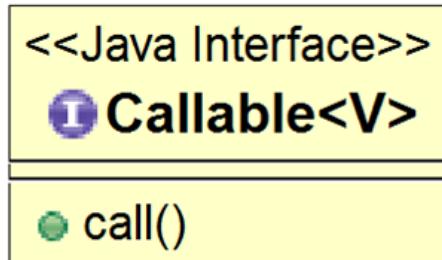
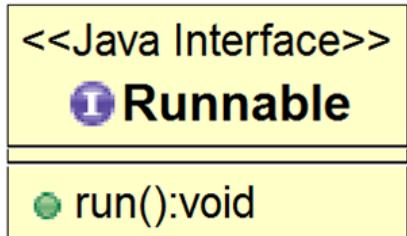


These features form the basis for Java streams & concurrency/parallelism frameworks

Overview of Functional Interfaces

Overview of Functional Interfaces

- A *functional interface* is an interface that contains only one abstract method



See www.oreilly.com/learning/java-8-functional-interfaces

Overview of Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument to a method

```
<T> void runTest(Function<T, T> fact, T n) {  
    long startTime = System.nanoTime();  
    T result = fact.apply(n);  
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;  
    ...  
}  
runTest(ParallelStreamFactorial::factorial, n);  
runTest(SequentialStreamFactorial::factorial, n);  
...
```

Overview of Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument to a method

```
<T> void runTest(Function<T, T> fact, T n) {  
    long startTime = System.nanoTime();  
    T result = fact.apply(n);  
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;  
    ...  
}  
  
runTest(ParallelStreamFactorial::factorial, n);  
runTest(SequentialStreamFactorial::factorial, n);  
...
```

*Record & print time taken
to compute 'n' factorial*

Overview of Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument to a method

```
<T> void runTest(Function<T, T> fact, T n)
    long startTime = System.nanoTime();
    T result = fact.apply(n);
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;
    ...
}

runTest(ParallelStreamFactorial::factorial, n);
runTest(SequentialStreamFactorial::factorial, n);
...
```

'fact' parameterizes the factorial implementation

Overview of Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument to a method

```
<T> void runTest(Function<T, T> fact, T n) {  
    long startTime = System.nanoTime();  
    T result = fact.apply(n);  
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;  
    ...  
}  
  
runTest(ParallelStreamFactorial::factorial, n);  
runTest(SequentialStreamFactorial::factorial, n);  
...
```

*Different factorial implementations can be passed as
method reference params to the runTest() method*

This is an example of behavior parameterization

Overview of Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument to a method

```
<T> void runTest(Function<T, T> fact, T n) {  
    long startTime = System.nanoTime();  
    T result = fact.apply(n);  
    long stopTime = (System.nanoTime() - startTime) / 1_000_000;  
    ...  
}  
runTest(ParallelStreamFactorial::factorial, n);  
...  
  
static BigInteger factorial(BigInteger n) {  
    return LongStream.rangeClosed(1, n)  
        .parallel()  
        .mapToObj(BigInteger::valueOf)  
        .reduce(BigInteger.ONE, BigInteger::multiply);  
}
```

Summary of Common Functional Interfaces

Summary of Common Functional Interfaces

- Java defines many types of functional interfaces

Package `java.util.function`

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
<code>BiConsumer<T,U></code>	Represents an operation that accepts two input arguments and returns no result.
<code>BiFunction<T,U,R></code>	Represents a function that accepts two arguments and produces a result.
<code>BinaryOperator<T></code>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
<code>BiPredicate<T,U></code>	Represents a predicate (boolean-valued function) of two arguments.
<code>BooleanSupplier</code>	Represents a supplier of boolean-valued results.
<code>Consumer<T></code>	Represents an operation that accepts a single input argument and returns no result.
<code>DoubleBinaryOperator</code>	Represents an operation upon two double-valued operands and producing a double-valued result.
<code>DoubleConsumer</code>	Represents an operation that accepts a single double-valued argument and returns no result.
<code>DoubleFunction<R></code>	Represents a function that accepts a double-valued argument and produces a result.
<code>DoublePredicate</code>	Represents a predicate (boolean-valued function) of one double-valued argument.
<code>DoubleSupplier</code>	Represents a supplier of double-valued results.
<code>DoubleToIntFunction</code>	Represents a function that accepts a double-valued argument and produces an int-valued result.
<code>DoubleToLongFunction</code>	Represents a function that accepts a double-valued argument and produces a long-valued result.
<code>DoubleUnaryOperator</code>	Represents an operation on a single double-valued operand that produces a double-valued result.
<code>Function<T,R></code>	Represents a function that accepts one argument and produces a result.

Summary of Common Functional Interfaces

- Java defines many types of functional interfaces
 - Some of these interfaces handle reference types

Package `java.util.function`

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
<code>BiConsumer<T,U></code>	Represents an operation that accepts two input arguments and returns no result.
<code>BiFunction<T,U,R></code>	Represents a function that accepts two arguments and produces a result.
<code>BinaryOperator<T></code>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
<code>BiPredicate<T,U></code>	Represents a predicate (boolean-valued function) of two arguments.
<code>BooleanSupplier</code>	Represents a supplier of boolean-valued results.
<code>Consumer<T></code>	Represents an operation that accepts a single input argument and returns no result.
<code>DoubleBinaryOperator</code>	Represents an operation upon two double-valued operands and producing a double-valued result.
<code>DoubleConsumer</code>	Represents an operation that accepts a single double-valued argument and returns no result.
<code>DoubleFunction<R></code>	Represents a function that accepts a double-valued argument and produces a result.
<code>DoublePredicate</code>	Represents a predicate (boolean-valued function) of one double-valued argument.
<code>DoubleSupplier</code>	Represents a supplier of double-valued results.
<code>DoubleToIntFunction</code>	Represents a function that accepts a double-valued argument and produces an int-valued result.
<code>DoubleToLongFunction</code>	Represents a function that accepts a double-valued argument and produces a long-valued result.
<code>DoubleUnaryOperator</code>	Represents an operation on a single double-valued operand that produces a double-valued result.
<code>Function<T,R></code>	Represents a function that accepts one argument and produces a result.

Summary of Common Functional Interfaces

- Java defines many types of functional interfaces
 - Some of these interfaces handle reference types
 - Other interfaces support primitive types

Package `java.util.function`

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
<code>IntConsumer</code>	Represents an operation that accepts a single int-valued argument and returns no result.
<code>IntFunction<R></code>	Represents a function that accepts an int-valued argument and produces a result.
<code>IntPredicate</code>	Represents a predicate (boolean-valued function) of one int-valued argument.
<code>IntSupplier</code>	Represents a supplier of int-valued results.
<code>IntToDoubleFunction</code>	Represents a function that accepts an int-valued argument and produces a double-valued result.
<code>IntToLongFunction</code>	Represents a function that accepts an int-valued argument and produces a long-valued result.
<code>IntUnaryOperator</code>	Represents an operation on a single int-valued operand that produces an int-valued result.
<code>LongBinaryOperator</code>	Represents an operation upon two long-valued operands and producing a long-valued result.
<code>LongConsumer</code>	Represents an operation that accepts a single long-valued argument and returns no result.
<code>LongFunction<R></code>	Represents a function that accepts a long-valued argument and produces a result.
<code>LongPredicate</code>	Represents a predicate (boolean-valued function) of one long-valued argument.
<code>LongSupplier</code>	Represents a supplier of long-valued results.
<code>LongToDoubleFunction</code>	Represents a function that accepts a long-valued argument and produces a double-valued result.
<code>LongToIntFunction</code>	Represents a function that accepts a long-valued argument and produces an int-valued result.
<code>LongUnaryOperator</code>	Represents an operation on a single long-valued operand that produces a long-valued result.
<code>ObjDoubleConsumer<T></code>	Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
<code>ObjIntConsumer<T></code>	Represents an operation that accepts an object-valued and a int-valued argument, and returns no result.

See docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Summary of Common Functional Interfaces

- Java defines many types of functional interfaces
 - Some of these interfaces handle reference types
 - Other interfaces support primitive types
 - Avoids “auto-boxing” overhead



Package `java.util.function`

Functional interfaces provide target types for lambda expressions and method references.

See: Description

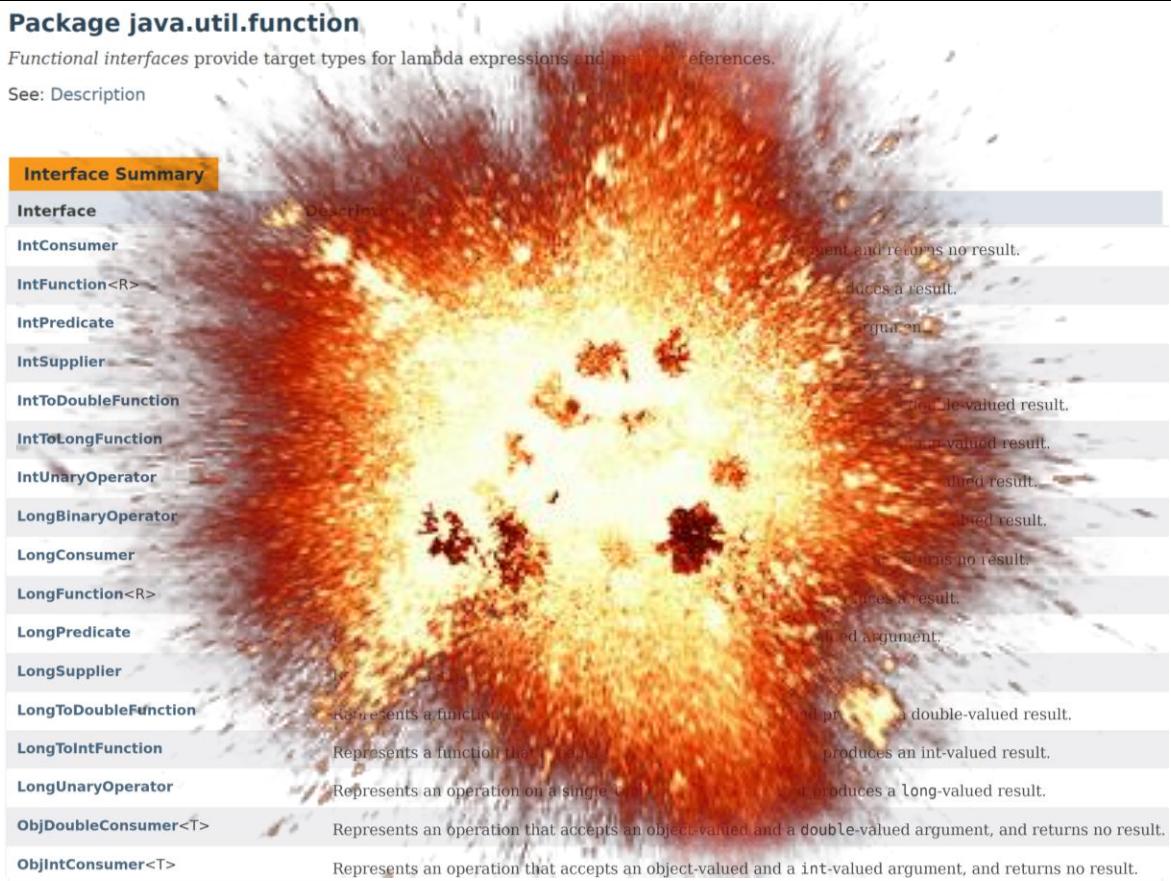
Interface Summary

Interface	Description
<code>IntConsumer</code>	Represents an operation that accepts a single int-valued argument and returns no result.
<code>IntFunction<R></code>	Represents a function that accepts an int-valued argument and produces a result.
<code>IntPredicate</code>	Represents a predicate (boolean-valued function) of one int-valued argument.
<code>IntSupplier</code>	Represents a supplier of int-valued results.
<code>IntToDoubleFunction</code>	Represents a function that accepts an int-valued argument and produces a double-valued result.
<code>IntToLongFunction</code>	Represents a function that accepts an int-valued argument and produces a long-valued result.
<code>IntUnaryOperator</code>	Represents an operation on a single int-valued operand that produces an int-valued result.
<code>LongBinaryOperator</code>	Represents an operation upon two long-valued operands and producing a long-valued result.
<code>LongConsumer</code>	Represents an operation that accepts a single long-valued argument and returns no result.
<code>LongFunction<R></code>	Represents a function that accepts a long-valued argument and produces a result.
<code>LongPredicate</code>	Represents a predicate (boolean-valued function) of one long-valued argument.
<code>LongSupplier</code>	Represents a supplier of long-valued results.
<code>LongToDoubleFunction</code>	Represents a function that accepts a long-valued argument and produces a double-valued result.
<code>LongToIntFunction</code>	Represents a function that accepts a long-valued argument and produces an int-valued result.
<code>LongUnaryOperator</code>	Represents an operation on a single long-valued operand that produces a long-valued result.
<code>ObjDoubleConsumer<T></code>	Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
<code>ObjIntConsumer<T></code>	Represents an operation that accepts an object-valued and a int-valued argument, and returns no result.

See rules.sonarsource.com/java/tag/performance/RSPEC-4276

Summary of Common Functional Interfaces

- Java defines many types of functional interfaces
 - Some of these interfaces handle reference types
 - Other interfaces support primitive types.
 - There's an explosion of Java functional interfaces!



Interface Summary	
Interface	Description
<code>IntConsumer</code>	Represents a function that consumes an int-valued argument and returns no result.
<code>IntFunction<R></code>	Represents a function that produces a result.
<code>IntPredicate</code>	Represents a predicate that takes an int-valued argument.
<code>IntSupplier</code>	Represents a supplier that produces an int-valued result.
<code>IntToDoubleFunction</code>	Represents a function that takes an int-valued argument and produces a double-valued result.
<code>IntToLongFunction</code>	Represents a function that takes an int-valued argument and produces a long-valued result.
<code>IntUnaryOperator</code>	Represents an operation on a single int-valued argument.
<code>LongBinaryOperator</code>	Represents an operation on two long-valued arguments.
<code>LongConsumer</code>	Represents a function that consumes a long-valued argument and returns no result.
<code>LongFunction<R></code>	Represents a function that produces a result.
<code>LongPredicate</code>	Represents a predicate that takes a long-valued argument.
<code>LongSupplier</code>	Represents a supplier that produces a long-valued result.
<code>LongToDoubleFunction</code>	Represents a function that takes a long-valued argument and produces a double-valued result.
<code>LongToIntFunction</code>	Represents a function that takes a long-valued argument and produces an int-valued result.
<code>LongUnaryOperator</code>	Represents an operation on a single long-valued argument.
<code>ObjDoubleConsumer<T></code>	Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
<code>ObjIntConsumer<T></code>	Represents an operation that accepts an object-valued and a int-valued argument, and returns no result.

See dzone.com/articles/whats-wrong-java-8-part-ii

Summary of Common Functional Interfaces

- Java defines many types of functional interfaces
 - Some of these interfaces handle reference types
 - Other interfaces support primitive types.
 - There's an explosion of Java functional interfaces!
 - However, learn these interfaces before trying to customize your own

Package `java.util.function`

Functional interfaces provide target types for lambda expressions and method references.

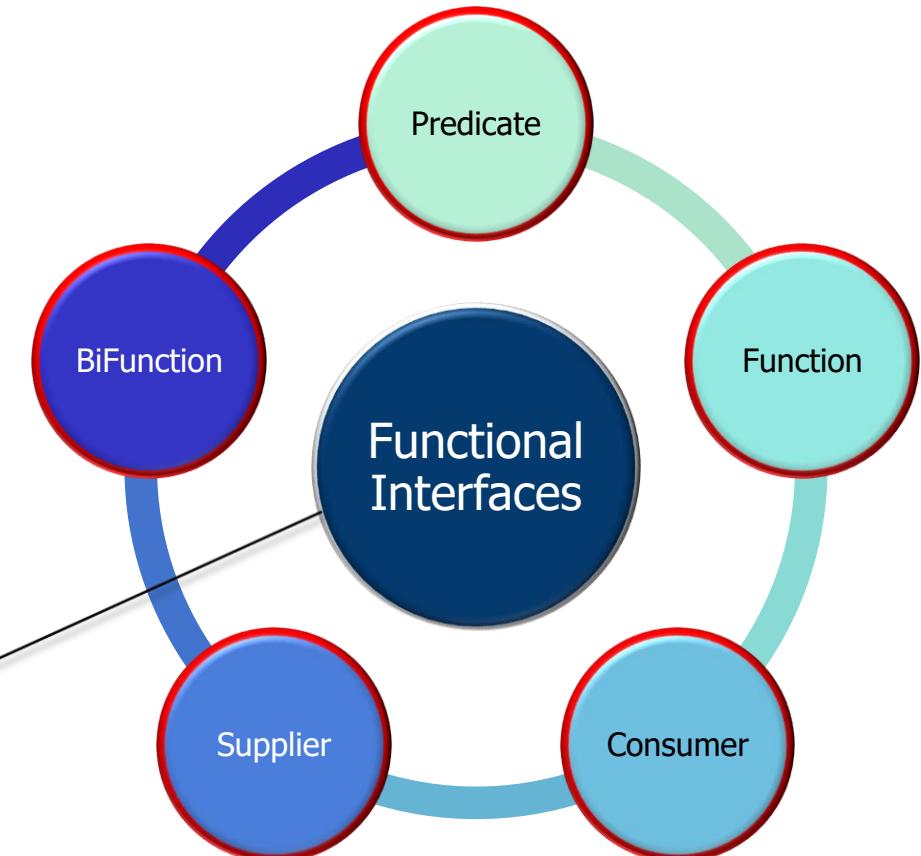
See: Description

Interface Summary

Interface	Description
<code>IntConsumer</code>	Represents an operation that accepts a single int-valued argument and returns no result.
<code>IntFunction<R></code>	Represents a function that accepts an int-valued argument and produces a result.
<code>IntPredicate</code>	Represents a predicate (boolean-valued function) of one int-valued argument.
<code>IntSupplier</code>	Represents a supplier of int-valued results.
<code>IntToDoubleFunction</code>	Represents a function that accepts an int-valued argument and produces a double-valued result.
<code>IntToLongFunction</code>	Represents a function that accepts an int-valued argument and produces a long-valued result.
<code>IntUnaryOperator</code>	Represents an operation on a single int-valued operand that produces an int-valued result.
<code>LongBinaryOperator</code>	Represents an operation upon two long-valued operands and producing a long-valued result.
<code>LongConsumer</code>	Represents an operation that accepts a single long-valued argument and returns no result.
<code>LongFunction<R></code>	Represents a function that accepts a long-valued argument and produces a result.
<code>LongPredicate</code>	Represents a predicate (boolean-valued function) of one long-valued argument.
<code>LongSupplier</code>	Represents a supplier of long-valued results.
<code>LongToDoubleFunction</code>	Represents a function that accepts a long-valued argument and produces a double-valued result.
<code>LongToIntFunction</code>	Represents a function that accepts a long-valued argument and produces an int-valued result.
<code>LongUnaryOperator</code>	Represents an operation on a single long-valued operand that produces a long-valued result.
<code>ObjDoubleConsumer<T></code>	Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
<code>ObjIntConsumer<T></code>	Represents an operation that accepts an object-valued and a int-valued argument, and returns no result.

Summary of Common Functional Interfaces

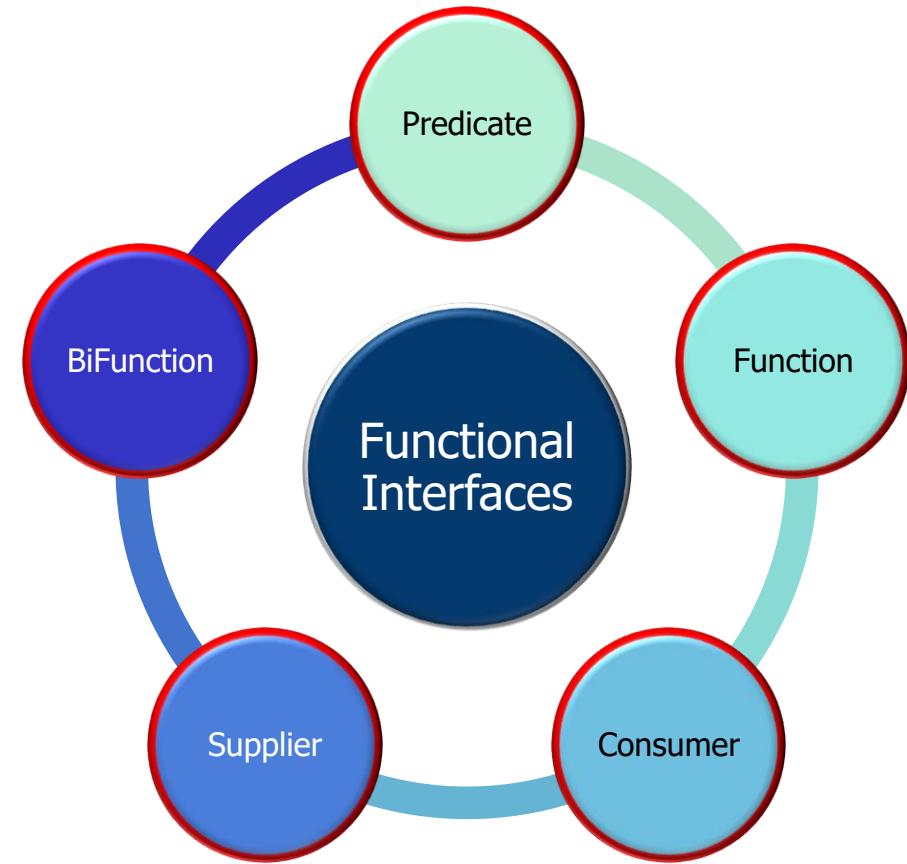
- Java defines many types of functional interfaces.
 - Some of these interfaces handle reference types.
 - Other interfaces support primitive types.
 - There's an explosion of Java functional interfaces!



We focus on the most common types of functional interfaces

Summary of Common Functional Interfaces

- Java defines many types of functional interfaces.
 - Some of these interfaces handle reference types.
 - Other interfaces support primitive types.
 - There's an explosion of Java functional interfaces!



All usages of functional interfaces in the upcoming examples are “stateless”!

End of Understand Java Functional Interfaces: Overview

Understand the Java Predicate Functional Interface

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

**Institute for Software
Integrated Systems**

**Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Part of the Lesson

- Recognize foundational functional programming features in Java, e.g.,
 - Lambda expressions
 - Method & constructor references
 - Key functional interfaces
 - Predicate

Interface Predicate<T>

Type Parameters:

T - the type of the input to the predicate

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface  
public interface Predicate<T>
```

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is `test(Object)`.

Learning Objectives in this Part of the Lesson

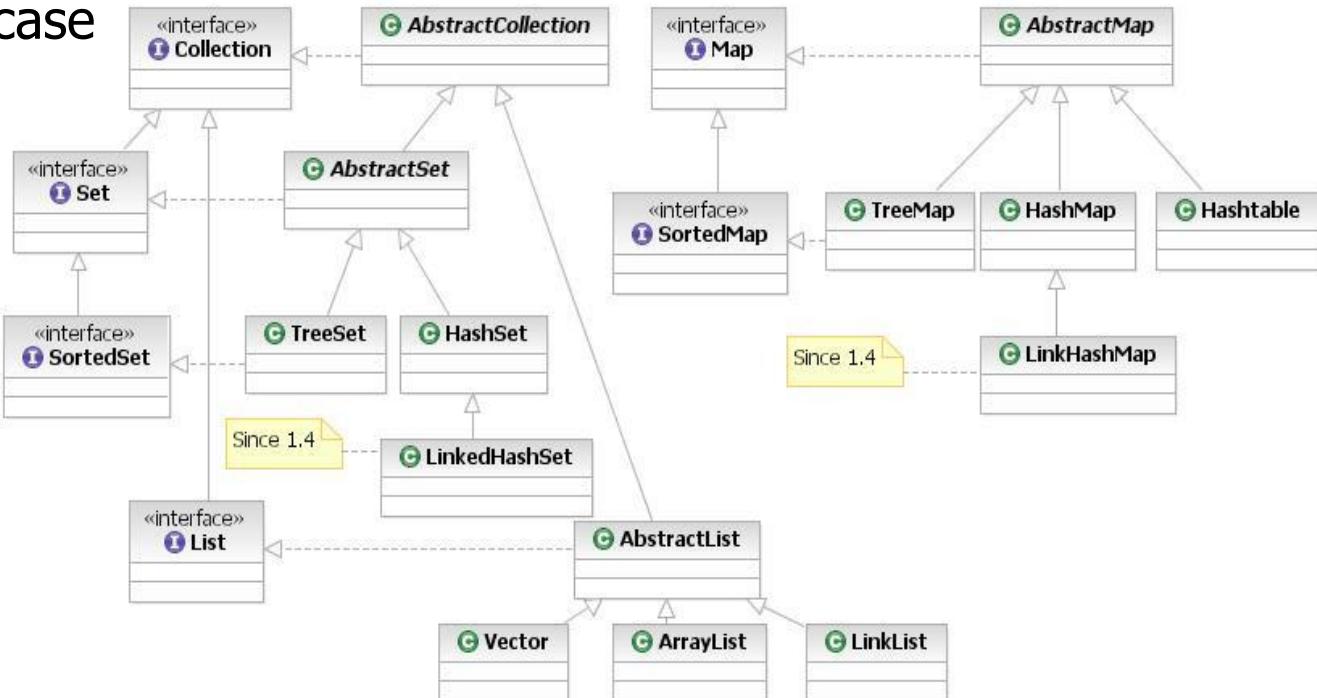
- Recognize foundational functional programming features in Java
- Understand how to apply Java predicates in concise example programs



See github.com/douglasraigschmidt/LiveLessons/tree/master/Java8

Learning Objectives in this Part of the Lesson

- Recognize foundational functional programming features in Java
- Understand how to apply Java predicates in concise example programs
 - The examples showcase the Java collections framework



See docs.oracle.com/javase/8/docs/technotes/guides/collections/

Overview of the Predicate Functional Interface

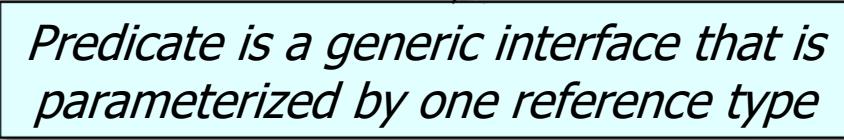
Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,
 - `public interface Predicate<T> { boolean test(T t); }`

Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

- ```
public interface Predicate<T> { boolean test(T t); }
```



*Predicate is a generic interface that is parameterized by one reference type*

# Overview of the Predicate Functional Interface

---

- A *Predicate* performs a test that returns true or false, e.g.,

- ```
public interface Predicate<T> { boolean test(T t) ; }
```



Its single abstract method is passed a parameter of type T & returns boolean

Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

- ```
public interface Predicate<T> { boolean test(T t); }
```



*The signature of the abstract method of the functional interface (called the "function descriptor") describes the signature of the lambda expression.*

# Overview of the Predicate Functional Interface

---

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }

Map<String, Integer> makeMap() {
 return new ConcurrentHashMap<String, Integer>() {
 put("Larry", 100); put("Curly", 90); put("Moe", 110);
 };
}

Map<String, Integer> iqMap = makeMap();

System.out.println(iqMap);

iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);

System.out.println(iqMap);
```

# Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }
```

```
Map<String, Integer> makeMap() {
 return new ConcurrentHashMap<String, Integer>() { {
 put("Larry", 100); | put("Curly", 90); put("Moe", 110);
 } };
}
```

*Create a map of "stooges" & their IQs!*

```
Map<String, Integer> iqMap = makeMap();
```

```
System.out.println(iqMap);
```

```
iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);
```

```
System.out.println(iqMap);
```



See [en.wikipedia.org/wiki/The\\_Three\\_Stooges](https://en.wikipedia.org/wiki/The_Three_Stooges)

# Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }

Map<String, Integer> makeMap() {
 return new ConcurrentHashMap<String, Integer>() {
 put("Larry", 100); put("Curly", 90); put("Moe", 110);
 });
}
```

```
Map<String, Integer> iqMap = makeMap();
System.out.println(iqMap);
```

```
iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);
```

```
System.out.println(iqMap);
```

*This predicate lambda removes all entries with iq <= 100*

See [docs.oracle.com/javase/8/docs/api/java/util/Collection.html#removeIf](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#removeIf)

# Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }
```

```
Map<String, Integer> makeMap() {
 return new ConcurrentHashMap<String, Integer>() { {
 put("Larry", 100); put("Curly", 90); put("Moe", 110);
 } };
}
```

```
Map<String, Integer> iqMap = makeMap();
```

```
System.out.println(iqMap);
```

```
iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);
```

```
System.out.println(iqMap);
```

This lambda implements  
the abstract test() method  
of Predicate directly inline

# Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }

Map<String, Integer> makeMap() {
 return new ConcurrentHashMap<String, Integer>() {
 put("Larry", 100); put("Curly", 90); put("Moe", 110);
 });
}
```

```
Map<String, Integer> iqMap = makeMap();
```

```
System.out.println(iqMap);
```

```
iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);
```

```
System.out.println(iqMap);
```

*entry* is short for (*Entry*  
*<String, Integer>* *entry*)  
via Java type inference

# Overview of the Predicate Functional Interface

---

- A *Predicate* performs a test that returns true or false, e.g.,

- ```
public interface Predicate<T> { boolean test(T t); }
```

```
interface Collection<E> {  
    ...  
    default boolean removeIf(Predicate<? super E> filter) {  
        ...  
        final Iterator<E> each = iterator();  
        while (each.hasNext()) {  
            if (filter.test(each.next())) {  
                each.remove();  
            }  
        }  
    }  
}
```

Here's how the `removeIf()` method uses the predicate passed to it

Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

- ```
public interface Predicate<T> { boolean test(T t); }
```

```
interface Collection<E> {
```

```
...
```

```
default boolean removeIf(Predicate<? super E> filter) {
 ...
 final Iterator<E> each = iterator();
 while (each.hasNext()) {
 if (filter.test(each.next())) {
 each.remove();
 }
 }
}
```

*Default methods enable adding new functionality to the interfaces of libraries & ensure binary compatibility with code written for older versions of those interfaces.*

See [docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html)

# Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }

interface Collection<E> {
 ...
 default boolean removeIf(Predicate<? super E> filter) {
 ...
 final Iterator<E> each = iterator();
 while (each.hasNext()) {
 if (filter.test(each.next())) {
 each.remove();
 }
 }
 }
}
```

'super' is a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type

# Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

- ```
public interface Predicate<T> { boolean test(T t); }
```

```
interface Collection<E> {
```

```
...
```

```
default boolean removeIf(Predicate<? super E> filter) {
```

```
...
```

```
final Iterator<E> each = iterator();
```

```
while (each.hasNext()) {
```

```
    if (filter.test(each.next())) {
```

```
        each.remove();
```

```
...
```

entry ->
`entry.getValue()`
`<= 100`

This predicate parameter is bound to the lambda expression passed to it

Overview of the Predicate Functional Interface

- A *Predicate* performs a test that returns true or false, e.g.,

```
public interface Predicate<T> { boolean test(T t); }

interface Collection<E> {
    ...
    default boolean removeIf(Predicate<? super E> filter) {
        ...
        final Iterator<E> each = iterator();
        while (each.hasNext()) {
            if (filter.test(each.next())) {
                each.remove();
            }
        }
    }
}
```



```
if (each.next().getValue() <= 100)
```

The 'entry' in the lambda predicate is replaced by the parameter to test()

Composing Predicates

Composing Predicates

- It's also possible to compose predicates.

```
• public interface Predicate<T> { boolean test(T t); }  
Map<String, Integer> iqMap = makeMap();
```

```
System.out.println(iqMap);
```

Create two predicate objects.

```
Predicate<ConcurrentMap.Entry<String, Integer>>  
entry -> entry.getValue() <= 100;
```

```
Predicate<ConcurrentMap.Entry<String, Integer>>  
entry -> entry.getKey().equals("Curly");
```

```
iqMap.entrySet().removeIf(lowIq.and(curly));
```

```
System.out.println(iqMap);
```

lowIq =

curly =



Composing Predicates

- It's also possible to compose predicates.

```
• public interface Predicate<T> { boolean test(T t); }
```

```
Map<String, Integer> iqMap = makeMap();
```

```
System.out.println(iqMap);
```

```
Predicate<ConcurrentMap.Entry<String, Integer>> lowIq =  
    entry -> entry.getValue() <= 100;
```

```
Predicate<ConcurrentMap.Entry<String, Integer>> curly =  
    entry -> entry.getKey().equals("Curly");
```

```
iqMap.entrySet().removeIf(lowIq.and(curly));
```

```
System.out.println(iqMap);
```



Compose two predicates!

End of Understand the Java Predicate Functional Interface

Understand the Java Function Functional Interface

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

**Institute for Software
Integrated Systems**

**Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Lesson

- Understand foundational functional programming features in Java, e.g.,
 - Lambda expressions
 - Method & constructor references
 - Key functional interfaces
 - Predicate
 - Function

Interface Function<T,R>

Type Parameters:

T - the type of the input to the function

R - the type of the result of the function

All Known Subinterfaces:

UnaryOperator<T>

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface  
public interface Function<T,R>
```

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is apply(Object).

Learning Objectives in this Part of the Lesson

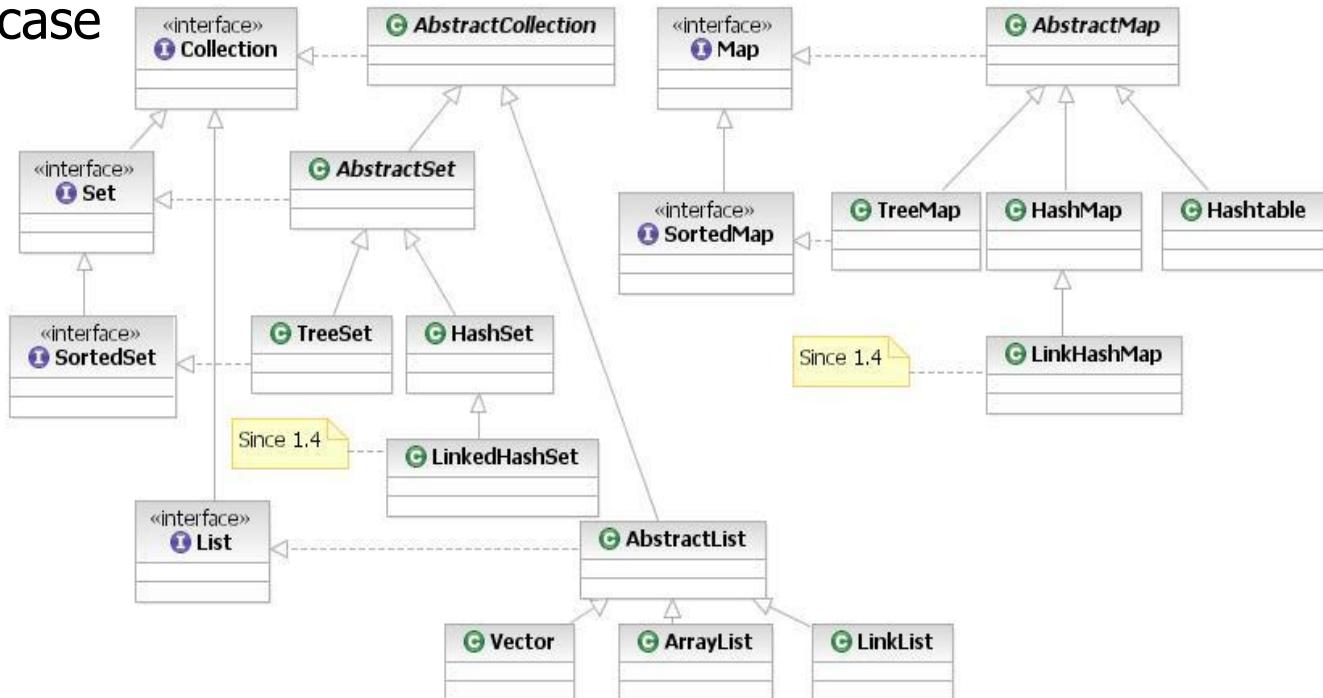
- Understand foundational functional programming features in Java
- Learn how to apply Java functions in concise example programs



See github.com/douglasraigschmidt/LiveLessons/tree/master/Java8

Learning Objectives in this Part of the Lesson

- Understand foundational functional programming features in Java
- Learn how to apply Java functions in concise example programs
 - The examples showcase the Java collections framework



See docs.oracle.com/javase/8/docs/technotes/guides/collections

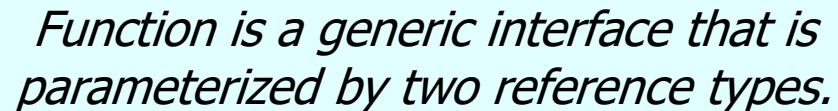
Overview of the Function Functional Interface

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
 - `public interface Function<T, R> { R apply(T t); }`

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
 - `public interface Function<T, R> { R apply(T t); }`



Function is a generic interface that is parameterized by two reference types.

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
 - `public interface Function<T, R> { R apply(T t); }`

*Its abstract method is passed a parameter
of type T & returns a value of type R.*

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =
 new ConcurrentHashMap<>();
```

*This map caches the results  
of prime # computations*

```
...
```

```
Long smallestFactor = primeCache.computeIfAbsent
(primeCandidate, (key) -> primeChecker(key));
```

```
...
```

```
Integer primeChecker(Integer primeCandidate) {
 ... // Returns 0 if a number is prime or the smallest
 // factor if it's not prime
}
```

# Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =  
    new ConcurrentHashMap<>();
```

*If key isn't already associated with a value, atomically compute
the value using the given mapping function & enter it into the map*

```
...
```

```
Long smallestFactor = primeCache.computeIfAbsent  
(primeCandidate, (key) -> primeChecker(key));
```

```
...
```

```
Integer primeChecker(Integer primeCandidate) {  
    ... // Returns 0 if a number is prime or the smallest  
        // factor if it's not prime  
}
```

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =
 new ConcurrentHashMap<>();
```

*This method provides atomic  
"check then act" semantics*

```
...
```

```
Long smallestFactor = primeCache.computeIfAbsent
(primeCandidate, (key) -> primeChecker(key));
```

```
...
```

```
Integer primeChecker(Integer primeCandidate) {
 ... // Returns 0 if a number is prime or the smallest
 // factor if it's not prime
}
```

# Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =  
    new ConcurrentHashMap<>();
```

A lambda expression that calls a function

```
...  
Long smallestFactor = primeCache.computeIfAbsent  
(primeCandidate, (key) -> primeChecker(key));  
...
```

```
Integer primeChecker(Integer primeCandidate) {  
    ... // Returns 0 if a number is prime or the smallest  
    // factor if it's not prime  
}
```

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

- ```
public interface Function<T, R> { R apply(T t); }
```

```
Map<Integer, Integer> primeCache =
 new ConcurrentHashMap<>();
```

*Could also be a passed as a method reference*

```
...
```

```
Long smallestFactor = primeCache.computeIfAbsent
(primeCandidate, this::primeChecker);
```

```
...
```

```
Integer primeChecker(Integer primeCandidate) {
 ... // Returns 0 if a number is prime or the smallest
 // factor if it's not prime
}
```

# Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
 public V computeIfAbsent(K key,
 Function<? super K, ? extends V> mappingFunction) {
 ...
 if ((f = tabAt(tab, i = (n - 1) & h)) == null)
 ...
 if ((val = mappingFunction.apply(key)) != null)
 node = new Node<K,V>(h, key, val, null);
 ...
}
```

Here's how computeIfAbsent() uses the function passed to it (atomically)

# Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
public V computeIfAbsent(K key,
 Function<? super K, ? extends V> mappingFunction) {
 ...
 if ((f = tabAt(tab, i = (n - 1) & h)) == null)
 ...
 if ((val = mappingFunction.apply(key)) != null)
 node = new Node<K,V>(h, key, val, null);
 ...
}
```

'super' is a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type

# Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
public V computeIfAbsent(K key,
 Function<? super K, ? extends V> mappingFunction) {
 ...
if ((f = tabAt(tab, i = (n - 1) & h)) == null)
 ...
if ((val = mappingFunction.apply(key)) != null)
 node = new Node<K,V>(h, key, val, null);
 ...
}
```

*'extends' is an upper bounded wildcard that restricts the unknown type to be a specific type or a subtype of that type*

# Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
public V computeIfAbsent(K key,
 Function<? super K, ? extends V> mappingFunction) {
```

'super' & 'extends' play different roles in Java generics

```
...
if ((f = tabAt(tab, i = (n - 1) & h)) == null)
 ...
if ((val = mappingFunction.apply(key)) != null)
 node = new Node<K,V>(h, key, val, null);
 ...
```

# Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
public V computeIfAbsent(K key,
 Function<? super K, ? extends V> mappingFunction) {
 ...
 if ((f = tabAt(tab, i = (n - 1) & h)) == null)
 ...
 if ((val = mappingFunction.apply(key)) != null)
 node = new Node<K,V>(h, key, val, null);
 ...
}
```

this::primeChecker

The function parameter is bound to this::primeChecker method reference

# Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,

```
• public interface Function<T, R> { R apply(T t); }

class ConcurrentHashMap<K,V> ...
public V computeIfAbsent(K key,
 Function<? super K, ? extends V> mappingFunction) {
```

```
 if ((val = primeChecker(key)) != null)
...
if ((f = tabAt(tab, i = (n - 1) & h)) == null)
...
if ((val = mappingFunction.apply(key)) != null)
 node = new Node<K,V>(h, key, val, null);
...
```

The `apply()` method is replaced with the `primeChecker()` lambda function

---

# Composing Functions

# Composing Functions

---

- It's also possible to compose functions.

- ```
public interface Function<T, R> { R apply(T t); }

class HtmlTagMaker {
    static String addLessThan(String t) { return "<" + t; }
    static String addGreaterThan(String t) { return t + ">"; }
}
```

```
Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
    .andThen(HtmlTagMaker::addGreaterThan);
```

```
System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
    + tagger.apply("/BODY") + tagger.apply("/HTML"));
```

Composing Functions

- It's also possible to compose functions.

```
• public interface Function<T, R> { R apply(T t); }

class HtmlTagMaker {
    static String addLessThan(String t) { return "<" + t; }
    static String /addGreaterThan(String t) { return t + ">"; }
}
```

These methods prepend '<' & append '>' to a string, respectively

```
Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
    .andThen(HtmlTagMaker::addGreaterThan);
```

```
System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
    + tagger.apply("/BODY") + tagger.apply("/HTML"));
```

Composing Functions

- It's also possible to compose functions.

- ```
public interface Function<T, R> { R apply(T t); }

class HtmlTagMaker {
 static String addLessThan(String t) { return "<" + t; }
 static String addGreaterThan(String t) { return t + ">"; }
}
```

*These functions prepend '<' & append '>' to a string*

```
Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
 .andThen(HtmlTagMaker::addGreaterThan);
```

```
System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
 + tagger.apply("/BODY") + tagger.apply("/HTML"))
```

# Composing Functions

- It's also possible to compose functions.

```
public interface Function<T, R> { R apply(T t); }

class HtmlTagMaker {
 static String addLessThan(String t) { return "<" + t; }
 static String addGreaterThan(String t) { return t + ">"; }
}
```

```
Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
 .andThen(HtmlTagMaker::addGreaterThan);
```



*This method composes two functions!*

```
System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
 + tagger.apply("/BODY") + tagger.apply("/HTML"));
```

# Composing Functions

- It's also possible to compose functions.

```
• public interface Function<T, R> { R apply(T t); }

class HtmlTagMaker {
 static String addLessThan(String t) { return "<" + t; }
 static String addGreaterThan(String t) { return t + ">"; }
}
```

```
Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
 .andThen(HtmlTagMaker::addGreaterThan);
```

*Prints "<HTML><BODY></BODY></HTML>"*

```
System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
 + tagger.apply("/BODY") + tagger.apply("/HTML"));
```

---

# End of Understand the Java Function Functional Interface

# **Understand the Java Supplier Functional Interface**

**Douglas C. Schmidt**

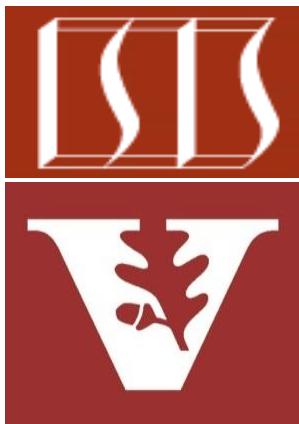
**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

---

- Understand foundational functional programming features in Java, e.g.,
  - Lambda expressions
  - Method & constructor references
  - Key functional interfaces
    - Predicate
    - Function
    - Supplier

## Interface Supplier<T>

### Type Parameters:

T - the type of results supplied by this supplier

### Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

---

```
@FunctionalInterface
public interface Supplier<T>
```

Represents a supplier of results.

There is no requirement that a new or distinct result be returned each time the supplier is invoked.

This is a functional interface whose functional method is `get()`.

# Learning Objectives in this Part of the Lesson

---

- Understand foundational functional programming features in Java
- Learn how to apply Java suppliers in concise example programs

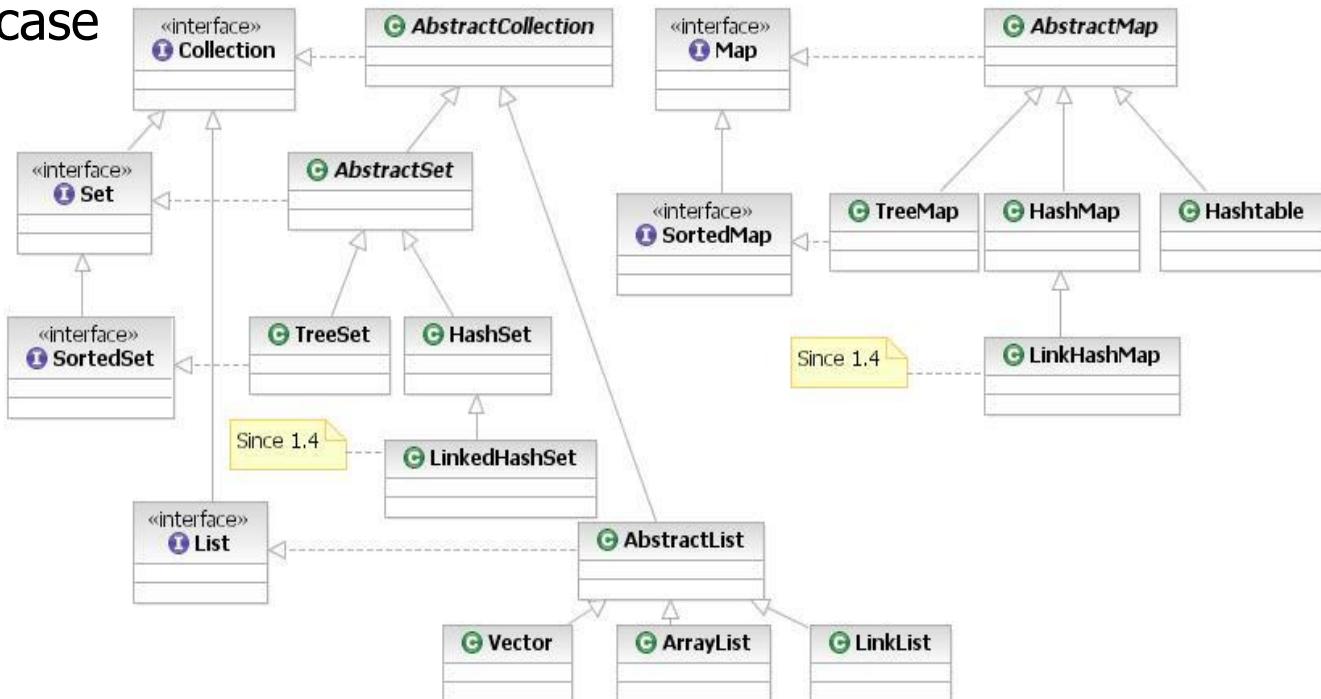


---

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8)

# Learning Objectives in this Part of the Lesson

- Understand foundational functional programming features in Java
- Learn how to apply Java suppliers in concise example programs
  - The examples showcase the Java collections framework



See [docs.oracle.com/javase/8/docs/technotes/guides/collections](https://docs.oracle.com/javase/8/docs/technotes/guides/collections)

---

# Overview of the Supplier Functional Interface

# Overview of Supplier Functional Interface

---

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

# Overview of Supplier Functional Interface

---

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`



*Supplier is a generic interface that is parameterized by one reference type*

# Overview of Supplier Functional Interface

---

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```



Its single abstract method is passed no parameters & returns a value of type T.

Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

```
Optional<String> disposition =
Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
+ being + " = "
+ disposition.orElseGet(() -> "unknown"));
```

# Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

```
• public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

Create a map associating each  
being with its personality traits

```
Optional<String> disposition =
Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
+ being + " = "
+ disposition.orElseGet(() -> "unknown"));
```

# Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

Get the name of a being from somewhere (e.g., prompt user)

```
Optional<String> disposition =
    Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
    + being + " = "
    + disposition.orElseGet(() -> "unknown"));
```

Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

*Return an optional describing the specified being  
if non-null, otherwise returns an empty Optional*

```
Optional<String> disposition =
 Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
 + being + " = "
 + disposition.orElseGet(() -> "unknown"));
```

# Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

```
• public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

```
Optional<String> disposition =
Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
+ being + " = "
+ disposition.orElseGet(() -> "unknown"));
```

A container object which may or  
may not contain a non-null value

# Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

```
Optional<String> disposition =
    Optional.ofNullable(beingMap.get(being));
```

Returns value if being is non-null

```
System.out.println("disposition of "
    + being + " = "
    + disposition.orElseGet(() -> "unknown"));
```

Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

```
Optional<String> disposition =
 Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
 + being + " = "
 + disposition.orElseGet(() -> "unknown"));
```

Returns supplier lambda  
value if being is not found

# Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };
```

```
String being = ...;
```

```
Optional<String> disposition =
Optional.ofNullable(beingMap.get(being));
```

```
System.out.println("disposition of "
+ being + " = "
+ disposition.orElse("unknown"));
```

Could also use orElse()

Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

- ```
public interface Supplier<T> { T get(); }
```

```
class Optional<T> {
```

```
 ...
```

```
 public T orElseGet(Supplier<? extends T> other) {
```

```
 return value != null
```

```
 ? value
```

```
 : other.get();
```

```
}
```

Here's how the `orElseGet()` method uses the supplier passed to it

# Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

```
• public interface Supplier<T> { T get(); }

class Optional<T> {
 ...
 public T orElseGet(Supplier<? extends T> other) {
 return value != null
 ? value
 : other.get();
 }
}
```

(`) -> "unknown"`

The string literal "unknown" is bound to the supplier lambda parameter

# Overview of Supplier Functional Interface

- A *Supplier* returns a value & takes no parameters, e.g.,

```
• public interface Supplier<T> { T get(); }

class Optional<T> {
 ...
 public T orElseGet(Supplier<? extends T> other) {
 return value != null
 ? value
 : other.get();
 }
}
```

`() -> "unknown"`

`"unknown"`

The string "unknown" is returned by orElseGet() if the value is null

---

# End of the Java Supplier Functional Interface

# **Understand the Java Consumer Functional Interface**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Lesson

---

- Understand foundational functional programming features in Java, e.g.,
  - Lambda expressions
  - Method & constructor references
  - Key functional interfaces
    - Predicate
    - Function
    - Supplier
    - Consumer

## Interface Consumer<T>

### Type Parameters:

T - the type of the input to the operation

### All Known Subinterfaces:

Stream.Builder<T>

### Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

# Learning Objectives in this Part of the Lesson

---

- Understand foundational functional programming features in Java
- Learn how to apply Java consumers in concise example programs

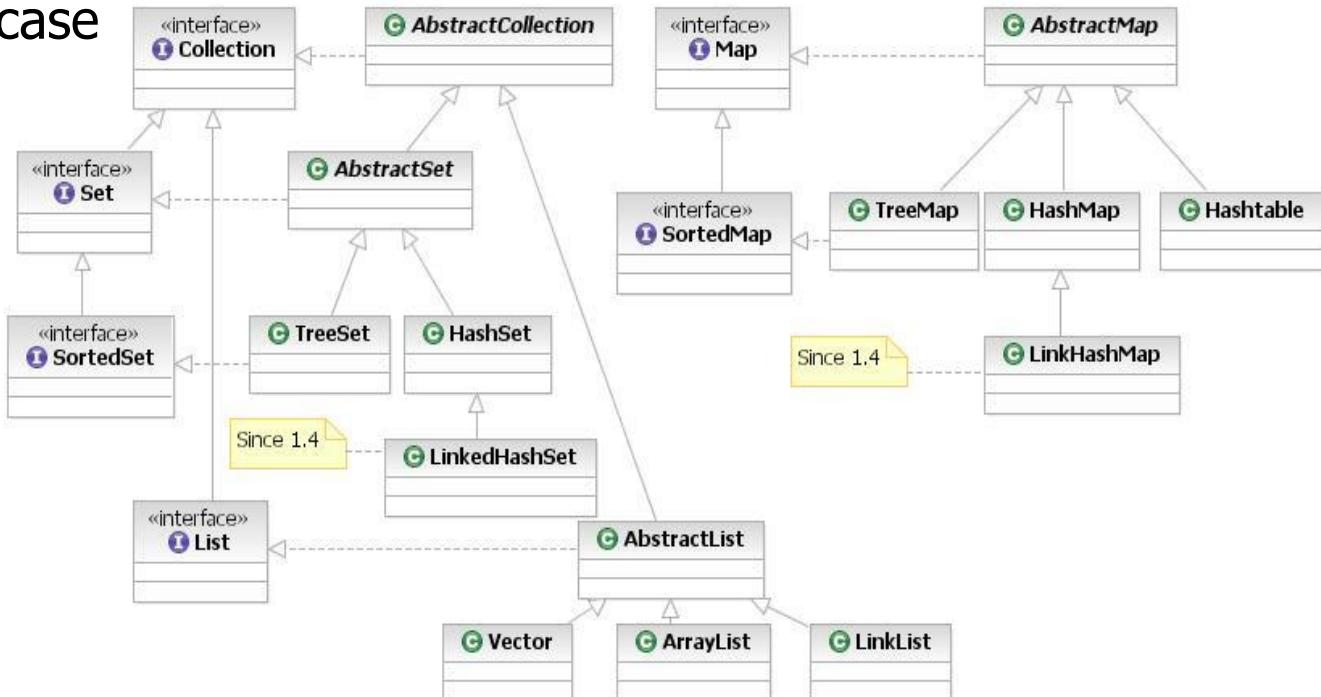


---

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8)

# Learning Objectives in this Part of the Lesson

- Understand foundational functional programming features in Java
- Learn how to apply Java consumers in concise example programs
  - The examples showcase the Java collections framework



See [docs.oracle.com/javase/8/docs/technotes/guides/collections/](https://docs.oracle.com/javase/8/docs/technotes/guides/collections/)

---

# Overview of the Consumer Functional Interface

# Overview of the Consumer Functional Interface

---

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`

# Overview of the Consumer Functional Interface

---

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`



*Consumer is a generic interface that is parameterized by one reference type*

# Overview of the Consumer Functional Interface

---

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`



*Its single abstract method is passed one parameter & returns nothing*

# Overview of the Consumer Functional Interface

- A *Consumer* accepts a parameter & returns no results, e.g.,

```
• public interface Consumer<T> { void accept(T t); }

List<Thread> threads = Arrays.asList(new Thread("Larry"),
 new Thread("Curly"),
 new Thread("Moe"));
```

*Create a list of threads with  
names of the three stooges*

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

# Overview of the Consumer Functional Interface

- A *Consumer* accepts a parameter & returns no results, e.g.,

- ```
public interface Consumer<T> { void accept(T t); }
```

```
List<Thread> threads = Arrays.asList(new Thread("Larry"),
                                         new Thread("Curly"),
                                         new Thread("Moe"));
```

Print out threads using forEach()

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

Overview of the Consumer Functional Interface

- A *Consumer* accepts a parameter & returns no results, e.g.,

- ```
public interface Consumer<T> { void accept(T t); }

public interface Iterable<T> {

 ...

 default void forEach(Consumer<? super T> action) {
 for (T t : this) {
 action.accept(t);
 }
 }
}
```

Here's how the `forEach()` method uses the `Consumer` passed to it

# Overview of the Consumer Functional Interface

- A *Consumer* accepts a parameter & returns no results, e.g.,

```
• public interface Consumer<T> { void accept(T t); }

public interface Iterable<T> {
 ...
 default void forEach(Consumer<? super T> action) {
 for (T t : this) {
 action.accept(t);
 }
 }
}
```

System.out::println

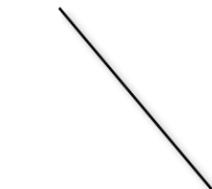
The consumer parameter is bound to the System.out::println method reference

# Overview of the Consumer Functional Interface

- A *Consumer* accepts a parameter & returns no results, e.g.,

```
• public interface Consumer<T> { void accept(T t); }

public interface Iterable<T> {
 ...
 default void forEach(Consumer<? super T> action) {
 for (T t : this) {
 action.accept(t);
 }
 }
}
```



```
System.out.println(t)
```

The accept() method is replaced by the call to System.out.println()

# Overview of the Consumer Functional Interface

- A *Consumer* accepts a parameter & returns no results, e.g.,

```
public interface Consumer<T> { void accept(T t); }

public interface Iterable<T> {
 ...
 default void forEach(Consumer<? super T> action) {
 for (T t : this) {
 action.accept(t);
 }
 }
}
```

*This use of "this" triggers the creation of  
the iterator associated with the subclass!!*

---

End of Understand  
the Consumer Java  
Functional Interface

# **Understand Other Properties of Java Functional Interfaces**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

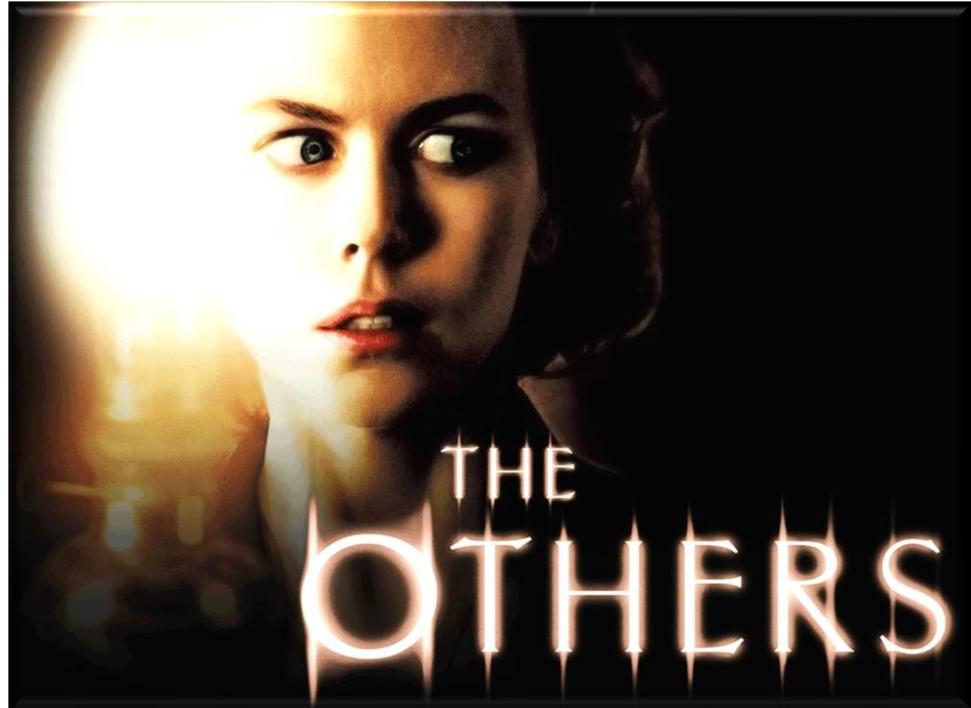
**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Lesson

---

- Understand foundational functional programming features in Java, e.g.,
  - Lambda expressions
  - Method & constructor references
  - Key functional interfaces
  - Other properties of functional interfaces



---

# Other Properties of Functional Interfaces

# Other Properties of Functional Interfaces

---

- Functional interfaces may also have default methods and/or static methods

```
#FunctionalInterface
interface Comparator<T> {
 int compare(T o1, T o2);

 boolean equals(Object obj);

 default Comparator<T> reversed()
 { return Collections.reverseOrder(this); }

 static <T extends Comparable<? super T>>
 Comparator<T> reverseOrder()
 { return Collections.reverseOrder(); }

 ...
}
```

# Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
#FunctionalInterface
interface Comparator<T> {
 int compare(T o1, T o2);

 boolean equals(Object obj);

 default Comparator<T> reversed()
 { return Collections.reverseOrder(this); }

 static <T extends Comparable<? super T>>
 Comparator<T> reverseOrder()
 { return Collections.reverseOrder(); }
 ...
```

*A comparison function that imposes a total ordering on some collection of objects*

# Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods

```
#FunctionalInterface
```

```
interface Comparator<T> {
 int compare(T o1, T o2);

 boolean equals(Object obj);

 default Comparator<T> reversed()
 { return Collections.reverseOrder(this); }

 static <T extends Comparable<? super T>>
 Comparator<T> reverseOrder()
 { return Collections.reverseOrder(); }

 ...
}
```

*This annotation type indicates that this interface type declaration is intended as a functional interface.*

# Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
#FunctionalInterface
interface Comparator<T> {
 int compare(T o1, T o2);
 boolean equals(Object obj);

 default Comparator<T> reversed()
 { return Collections.reverseOrder(this); }

 static <T extends Comparable<? super T>>
 Comparator<T> reverseOrder()
 { return Collections.reverseOrder(); }
 ...
}
```

*An abstract method in this functional interface*

# Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
#FunctionalInterface
interface Comparator<T> {
 int compare(T o1, T o2);
 boolean equals(Object obj);

 default Comparator<T> reversed()
 { return Collections.reverseOrder(this); }

 static <T extends Comparable<? super T>>
 Comparator<T> reverseOrder()
 { return Collections.reverseOrder(); }
 ...
}
```

*A default method provides the default implementation, which can be overridden*

# Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
#FunctionalInterface
interface Comparator<T> {
 int compare(T o1, T o2);

 boolean equals(Object obj);

 default Comparator<T> reversed()
 { return Collections.reverseOrder(this); }

 static <T extends Comparable<? super T>>
 Comparator<T> reverseOrder()
 { return Collections.reverseOrder(); }
 ...
}
```

*A static method provides the one-&-only implementation*

# Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.

```
#FunctionalInterface
interface Comparator<T> {
 int compare(T o1, T o2);

 boolean equals(Object obj);

 default Comparator<T> reversed()
 { return Collections.reverseOrder(this); }

 static <T extends Comparable<? super T>>
 Comparator<T> reverseOrder()
 { return Collections.reverseOrder(); }

 ...
}
```

*An abstract method that overrides a public java.lang.Object method does not count as part of the interface's abstract method count*

---

# End of Understand Other Properties of Java Functional Interfaces

# **Understand Java Streams: Overview**

**Douglas C. Schmidt**

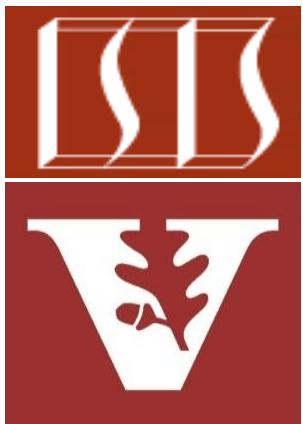
**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

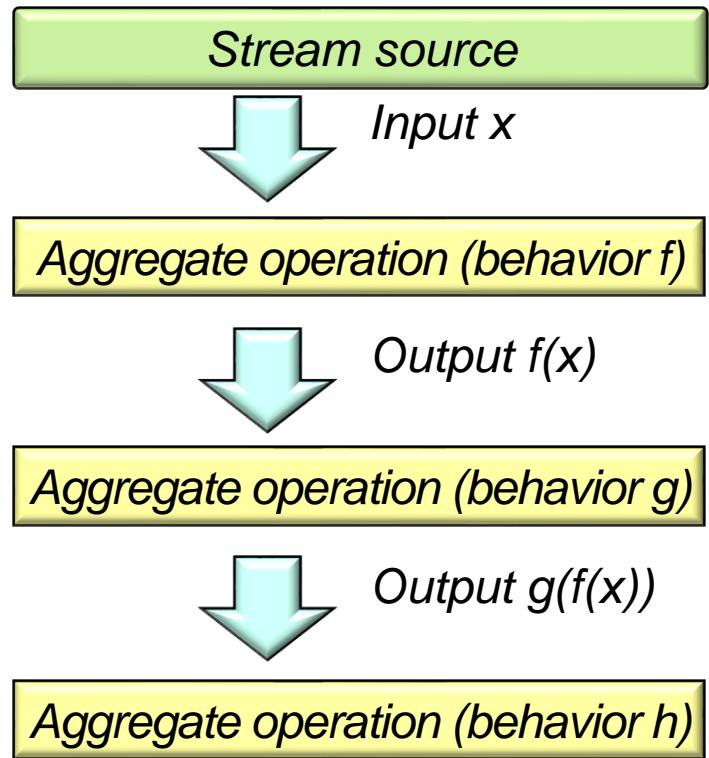
**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

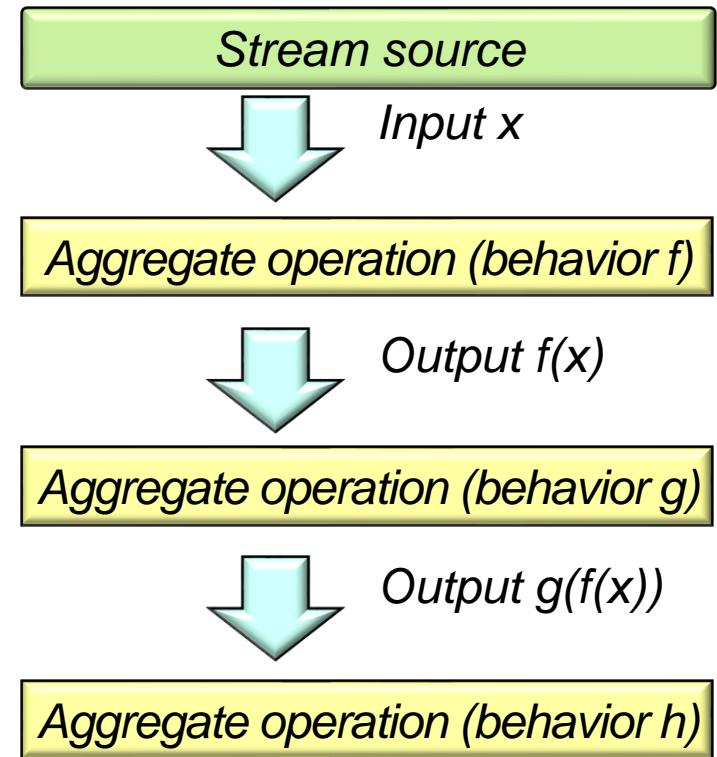
---

- Understand Java streams structure & functionality



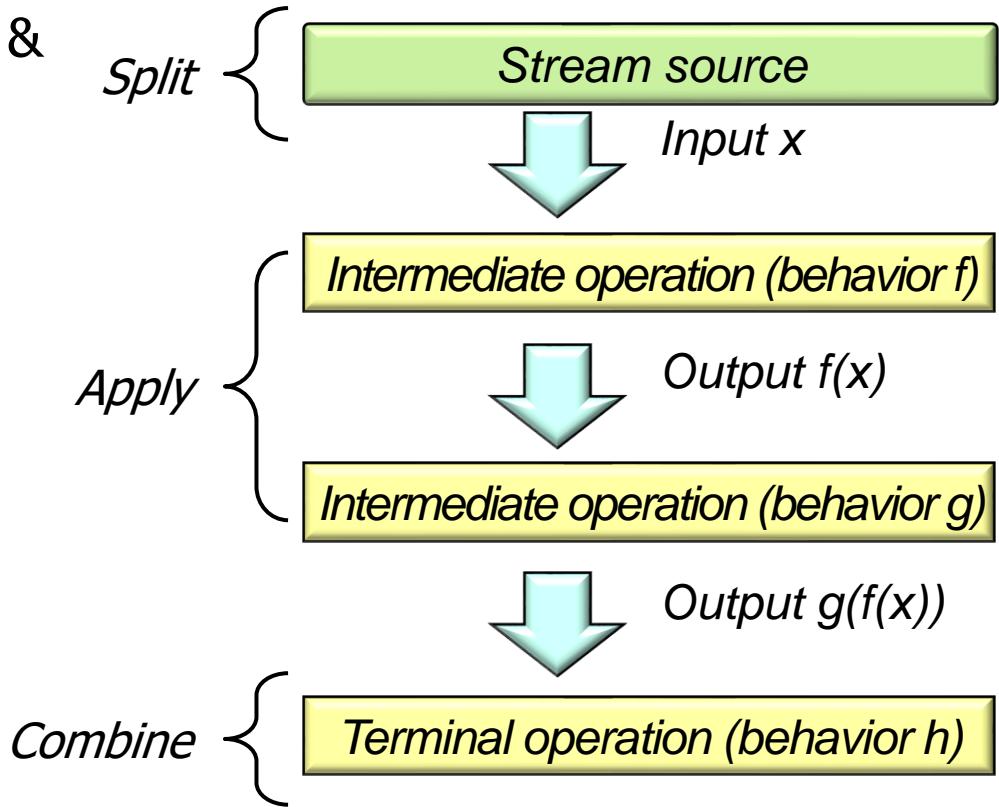
# Learning Objectives in this Part of the Lesson

- Understand Java streams structure & functionality, e.g.
  - Fundamentals of streams



# Learning Objectives in this Part of the Lesson

- Understand Java streams structure & functionality, e.g.
  - Fundamentals of streams
  - Three streams phases



---

# Overview of Java Streams

# Overview of Java Streams

- Java streams are a framework first introduced into the Java class library in Java 8



## What's New in JDK 8

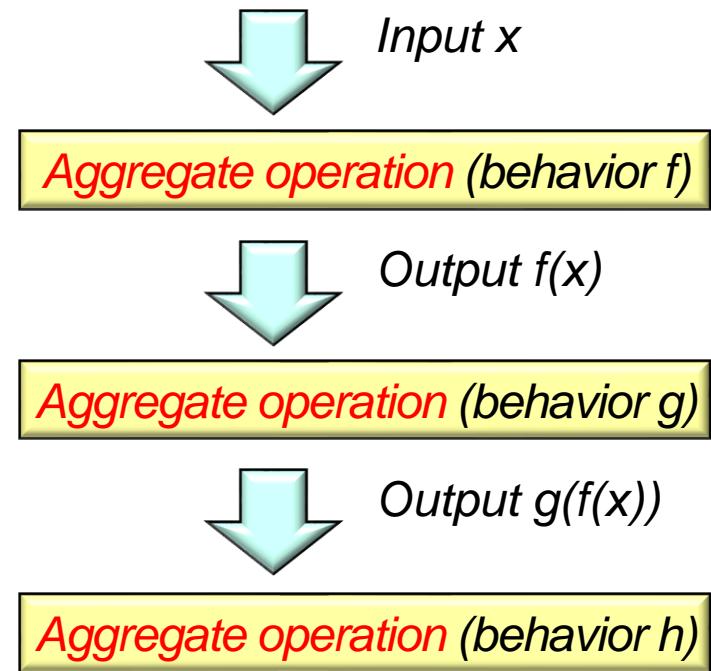
Java Platform, Standard Edition 8 is a major feature release. This document summarizes features and enhancements in Java SE 8 and in JDK 8, Oracle's implementation of Java SE 8. Click the component name for a more detailed description of the enhancements for that component.

- Java Programming Language
  - Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
  - Method references provide easy-to-read lambda expressions for methods that already have a name.
  - Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
  - Repeating Annotations provide the ability to apply the same annotation type more than once to the same declaration or type use.
  - Type Annotations provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
  - Improved type inference.
  - Method parameter reflection.
- Collections
  - Classes in the new `java.util.stream` package provide a Stream API to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API, which enables bulk operations on collections, such as sequential or parallel map-reduce transformations.
  - Performance Improvement for HashMaps with Key Collisions

See [docs.oracle.com/javase/tutorial/collectionsstreams/](https://docs.oracle.com/javase/tutorial/collectionsstreams/)

# Overview of Java Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values” or “data”)



See [docs.oracle.com/javase/tutorial/collections/streams](https://docs.oracle.com/javase/tutorial/collections/streams)

# Overview of Java Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, "values" or "data")



*An aggregate operation is a higher-order function that applies a "behavior" param to every element in a stream.*



*Aggregate operation (behavior f)*



*Aggregate operation (behavior g)*



*Aggregate operation (behavior h)*

# Overview of Java Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values” or “data”)



*Behavior parameterization simplifies coping with changing requirements.*



Aggregate operation (*behavior f*)



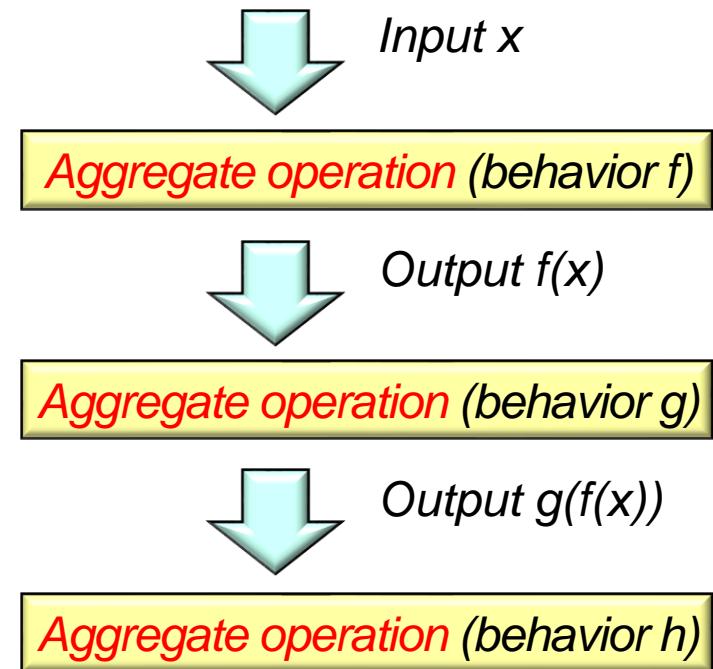
Aggregate operation (*behavior g*)



Aggregate operation (*behavior h*)

# Overview of Java Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values” or “data”)



*A stream is conceptually unbounded, though it's often bounded by practical constraints.*

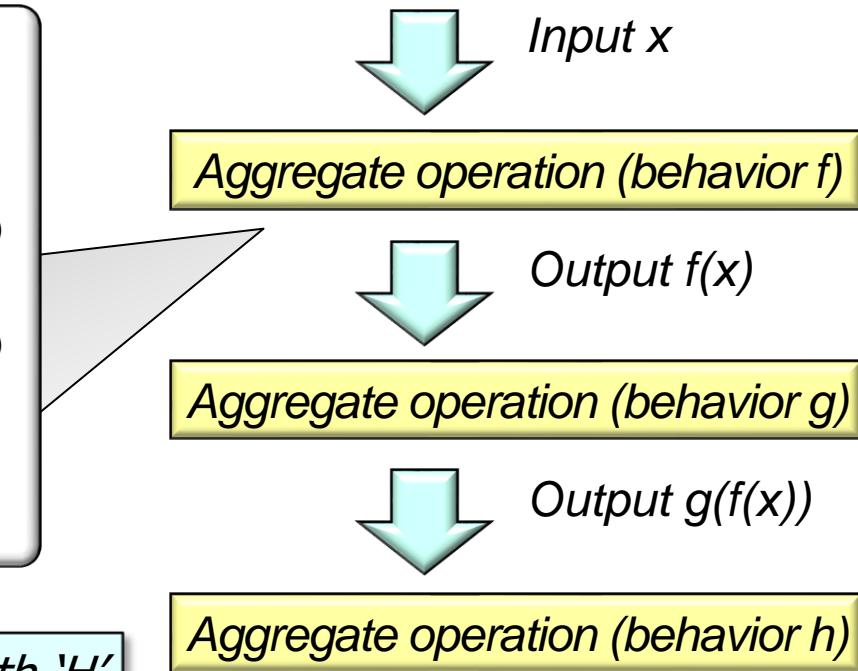
# Overview of Java Streams

- We use this stream as a case study example throughout this introduction

Stream

```
.of ("Ophelia", "horatio",
 "laertes", "Gertrude",
 "Hamlet", "fortinbras", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```

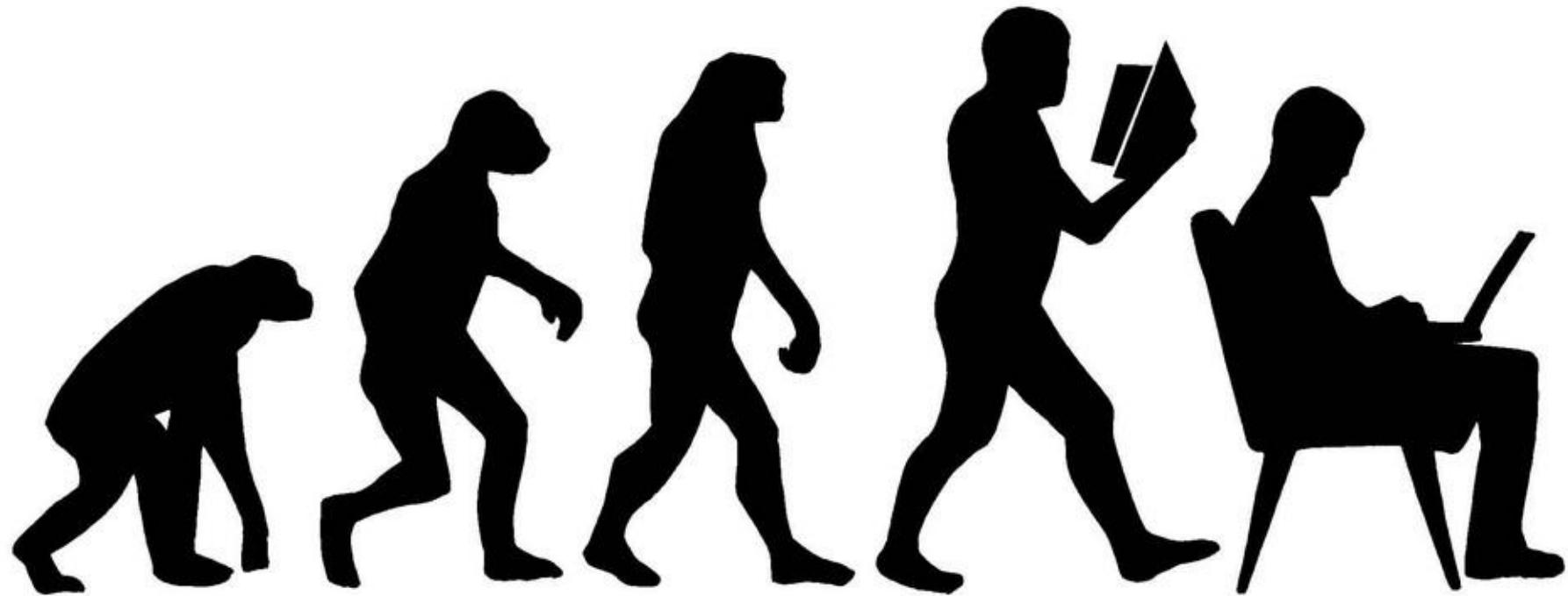
*Print each character in Hamlet that starts with 'H' or 'h' in consistently capitalized & sorted order.*



# Overview of Java Streams

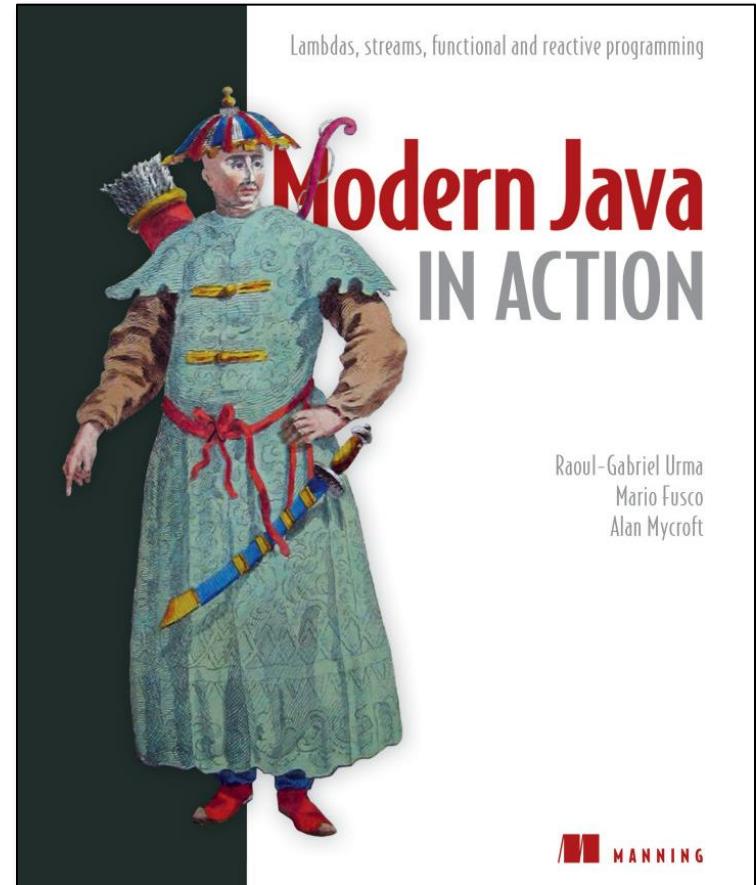
---

- Java streams have evolved a bit over time



# Overview of Java Streams

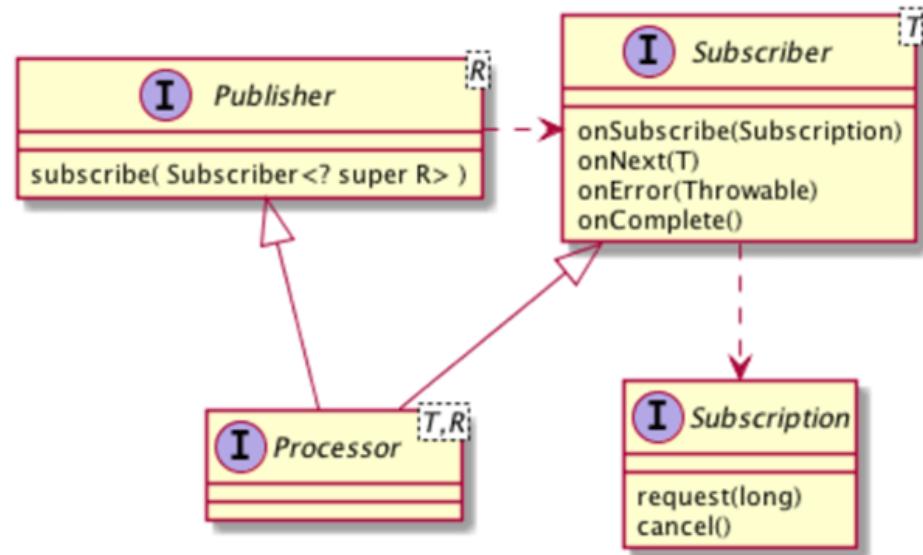
- Java streams have evolved a bit over time
  - e.g., Later versions of Java added some new operations



See [www.baeldung.com/java-9-stream-api](http://www.baeldung.com/java-9-stream-api) & [blog.codefx.org/java/teeing-collector](http://blog.codefx.org/java/teeing-collector)

# Overview of Java Streams

- Java 9 also added a new API that implements the reactive streams specification



# Overview of Java Streams

---

- Java 9 also added a new API that implements the reactive streams specification
  - Reactive streams is covered later in this course



Project  
Reactor

---

See upcoming lessons on RxJava & Project Reactor

---

# Overview of Stream Phases

# Overview of Stream Phases

---

- Streams usually have three phases



---

See [www.jstatsoft.org/article/view/v040i01/v40i01.pdf](http://www.jstatsoft.org/article/view/v040i01/v40i01.pdf)

# Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data



**Stream**

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
...
```

e.g., a Java **array**, collection, generator function, or input channel

# Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data



```
List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);
```

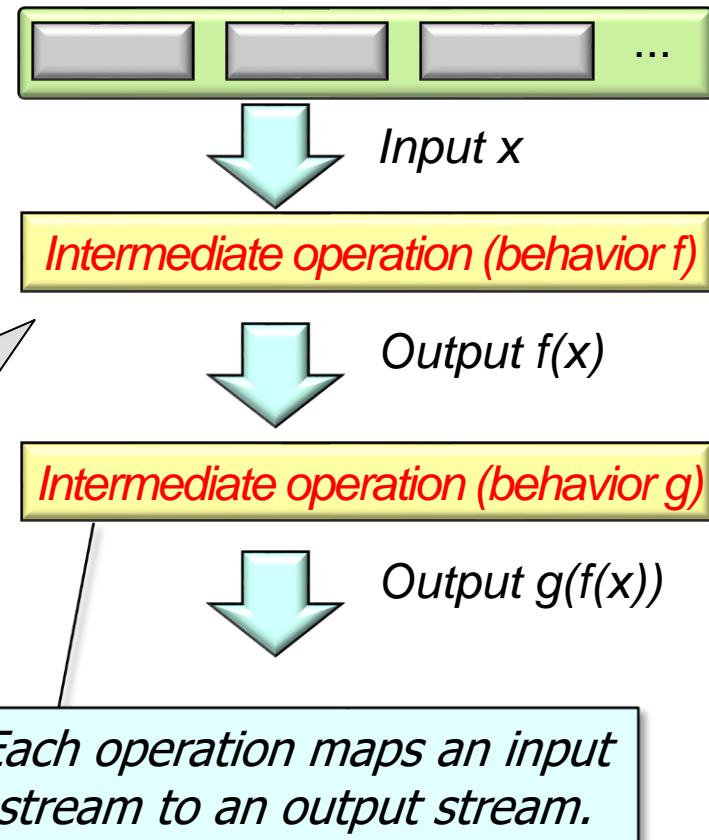
```
characters
.stream()
...
```

e.g., a Java array, collection, generator function, or input channel

# Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations

```
Stream
.of("horatio", "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
...
```



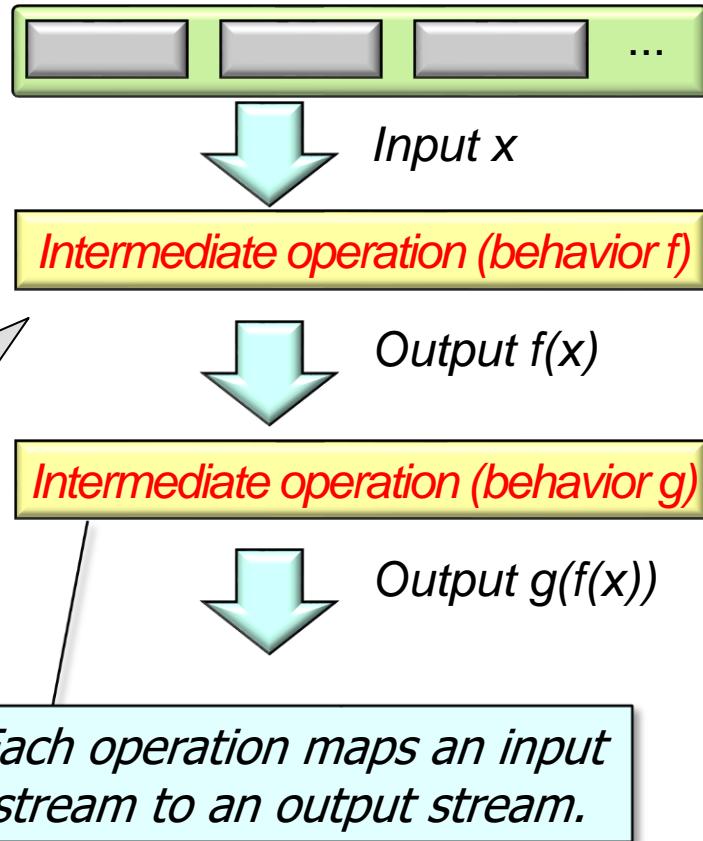
Examples of intermediate operations include filter(), map(), & sorted()

# Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
    - Processing often involves transforming

**Stream**

```
.of ("horatio", "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
...
```

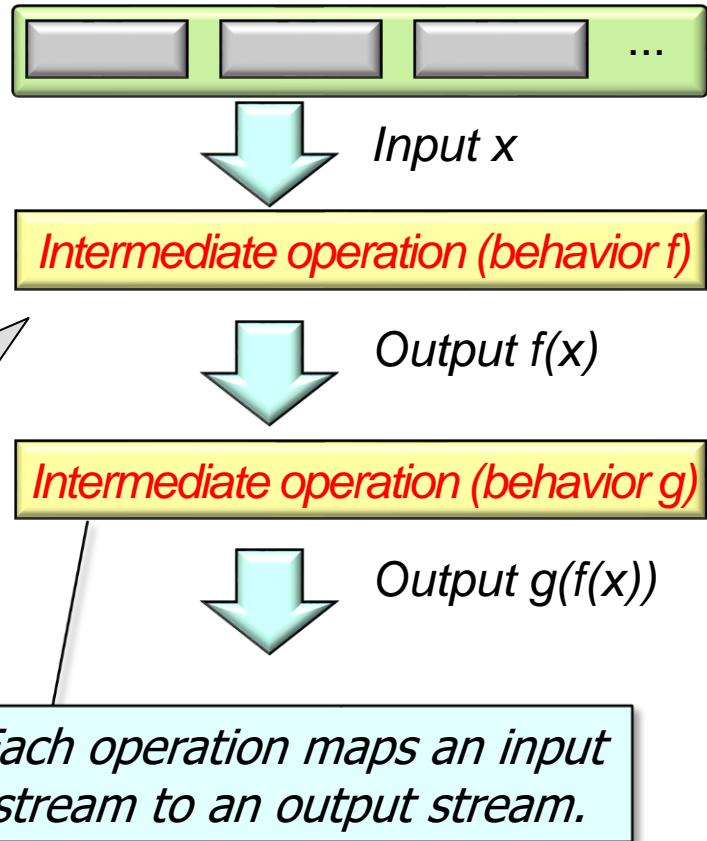


# Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
    - Processing often involves transforming

**Stream**

```
.of ("horatio", "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
...
```



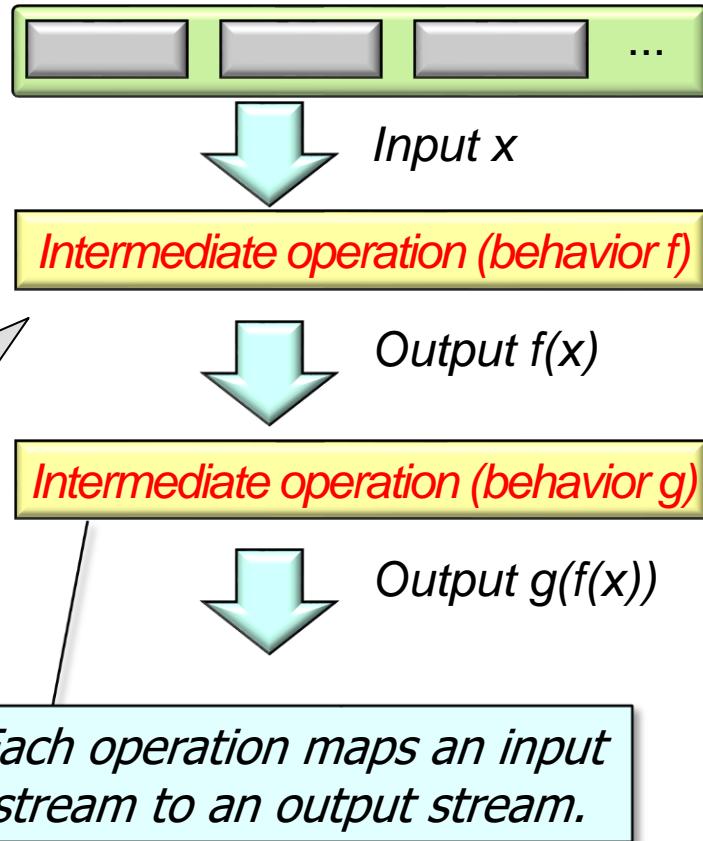
Some transformations are stateless & some are stateful

# Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
    - Processing often involves transforming

**Stream**

```
.of ("horatio", "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
...
```

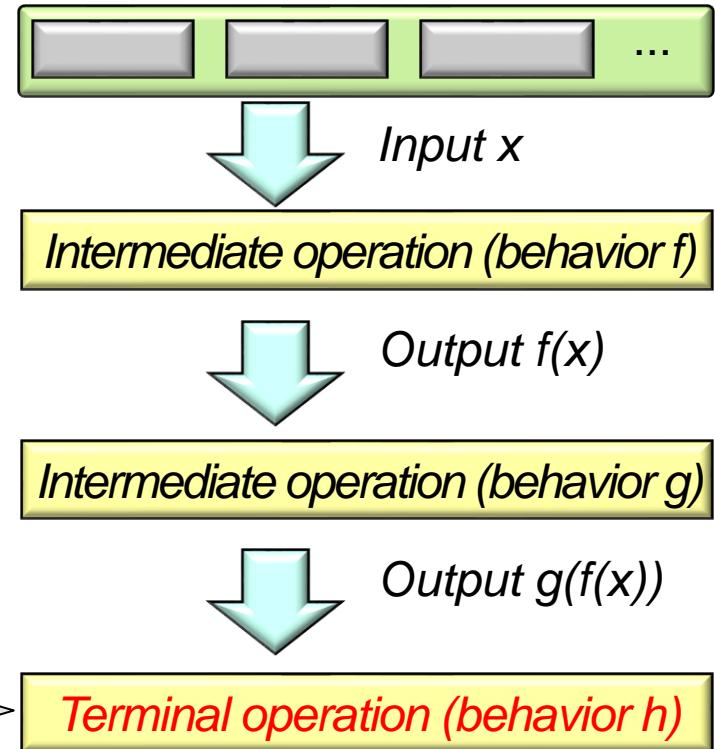


Some transformations are stateless & some are stateful

# Overview of Stream Phases

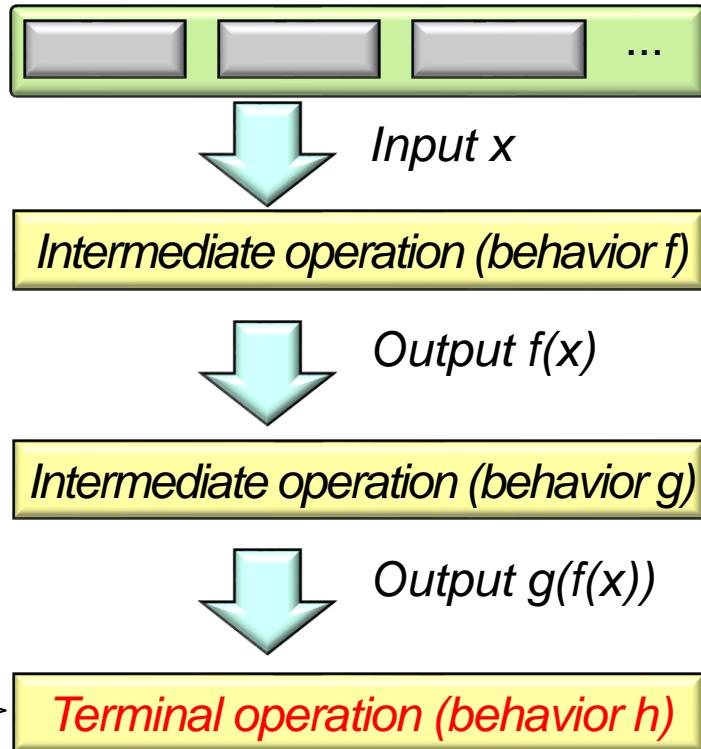
- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
  - **Combine** – finish with a terminal operation that yields a non-stream result

```
...
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



# Overview of Stream Phases

- Streams usually have three phases, i.e.
  - Split** – start with a source of data
  - Apply** – process data through a pipeline of intermediate operations
  - Combine** – finish with a terminal operation that yields a non-stream result



```
...
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```

A terminal operation triggers processing of intermediate operations in a stream

# Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
  - **Combine** – finish with a terminal operation that yields a non-stream result



A stream only runs if it has one (& only one) terminal operation

---

# End of Understand Java Streams: Overview

# **Understand Java Streams**

## **Common Operations**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

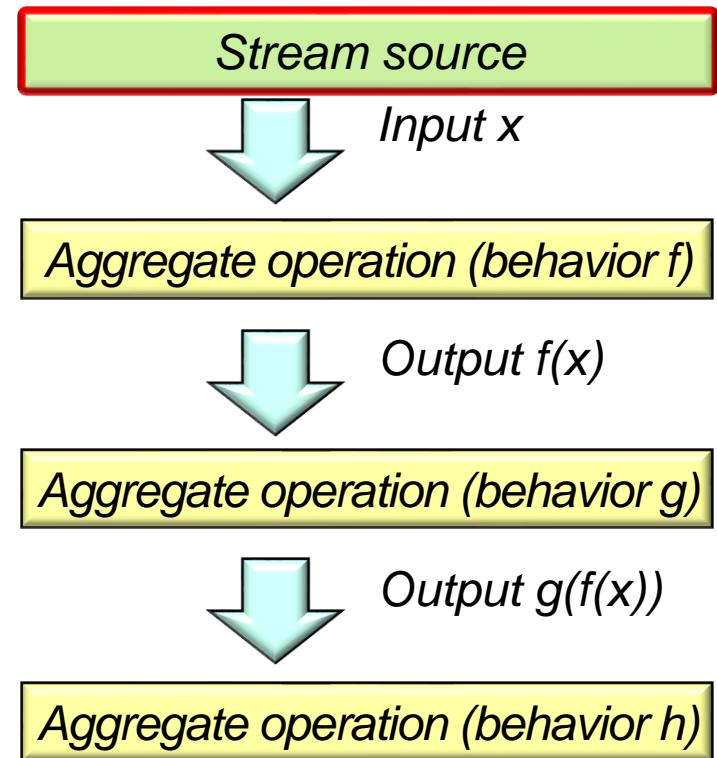
**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

---

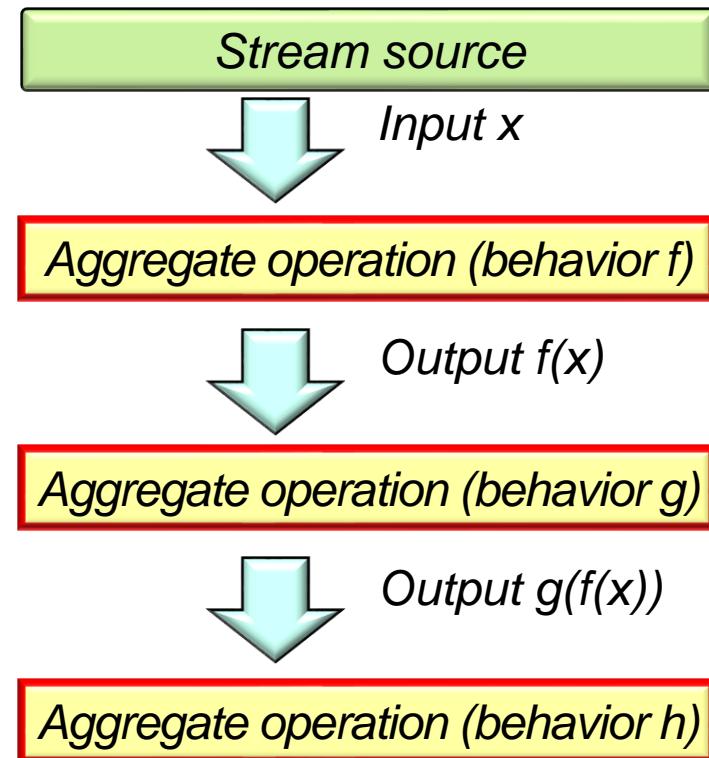
- Understand Java streams structure & functionality, e.g.
  - Fundamentals of streams
  - Three streams phases
  - Operations that create a stream



# Learning Objectives in this Part of the Lesson

---

- Understand Java streams structure & functionality, e.g.
  - Fundamentals of streams
  - Three streams phases
  - Operations that create a stream
  - Aggregate operations in a stream



---

# Operations that Create a Java Stream

# Operations that Create a Java Stream

- A factory method creates a stream from some source

**Stream**

```
.of ("horatio",
 "laertes",
 "Hamlet", ...)
```

...

*Stream source*

*Input x*

*Aggregate operation (behavior f)*

*Output f(x)*

*Aggregate operation (behavior g)*

*Output g(f(x))*

*Aggregate operation (behavior h)*

# Operations that Create a Java Stream

- A factory method creates a stream from some source

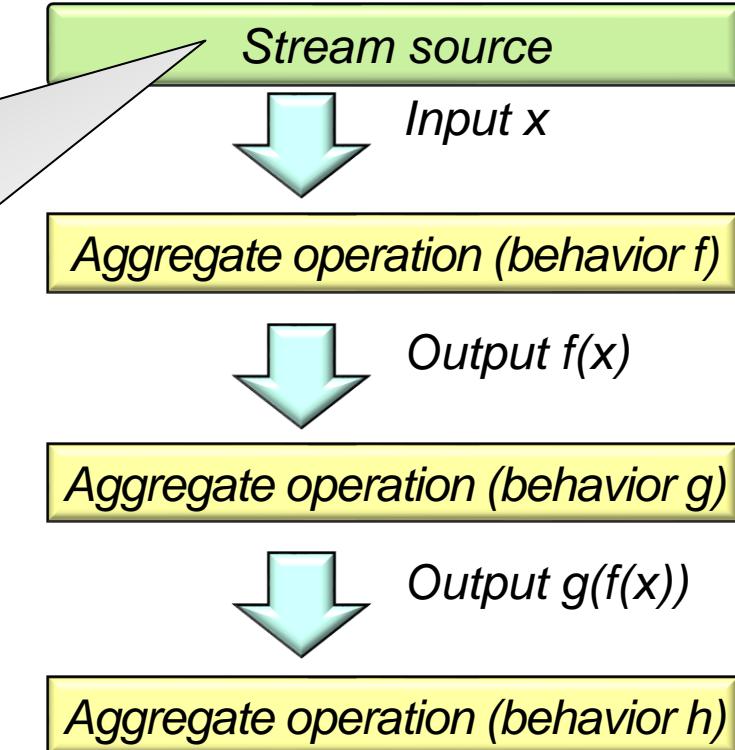
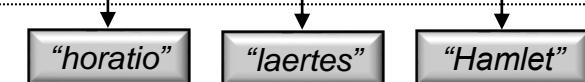
**Stream**

```
.of ("horatio",
 "laertes",
 "Hamlet", ...) ...
```

Array  
<String>



Stream  
<String>



*The Stream.of() factory method converts an array of T into a stream of T*

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#of](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#of)

# Operations that Create a Java Stream

---

- Many factory methods create streams

```
collection.stream()
```

```
collection.parallelStream()
```

```
Pattern.compile(...).splitAsStream()
```

```
Stream.of(value1,... ,valueN)
```

```
StreamSupport
```

```
 .stream(iterable.splitter(),
 false)
```

```
...
```



```
Arrays.stream(array)
```

```
Arrays.stream(array, start, end)
```

```
Files.lines(file_path)
```

```
"string".chars()
```

```
Stream.iterate(init_value,
 generate_expression)
```

```
Stream.builder().add(...).build()
```

```
Stream.generate(supplier)
```

```
Files.list(file_path)
```

```
Files.find(file_path, max_depth,
 matcher)
```

```
...
```

# Operations that Create a Java Stream

- Many factory methods create streams

```
collection.stream()
collection.parallelStream()
Pattern.compile(...).splitAsStream()
Stream.of(value1, ... ,valueN)
StreamSupport
 .stream(iterablespliterator(),
 false)
...
```

```
Arrays.stream(array)
Arrays.stream(array, start, end)
Files.lines(file_path)
"string".chars()
Stream.iterate(init_value,
 generate_expression)
Stream.builder().add(...).build()
Stream.generate(supplier)
Files.list(file_path)
Files.find(file_path, max_depth,
 matcher)
...
```

*These are key factory methods  
that we focus on in this course.*

---

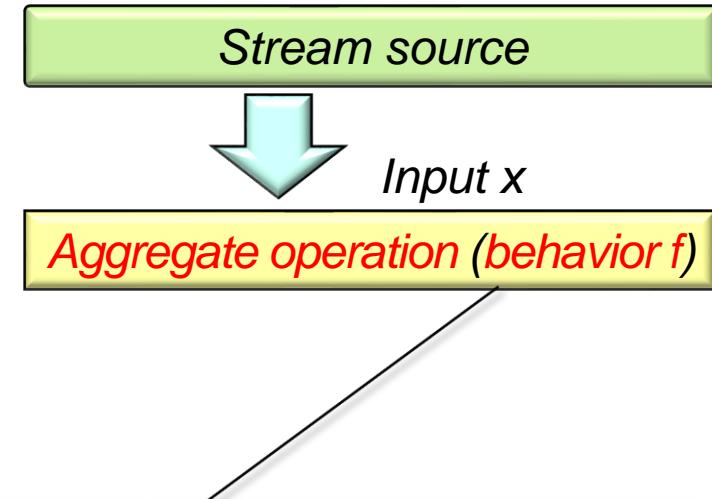
# Java Streams

# Aggregate Operations

# Java Streams Aggregate Operations

- An aggregate operation performs a *behavior* on elements in a stream

λ



*A behavior is implemented by a lambda expression or method reference corresponding to a functional interface*

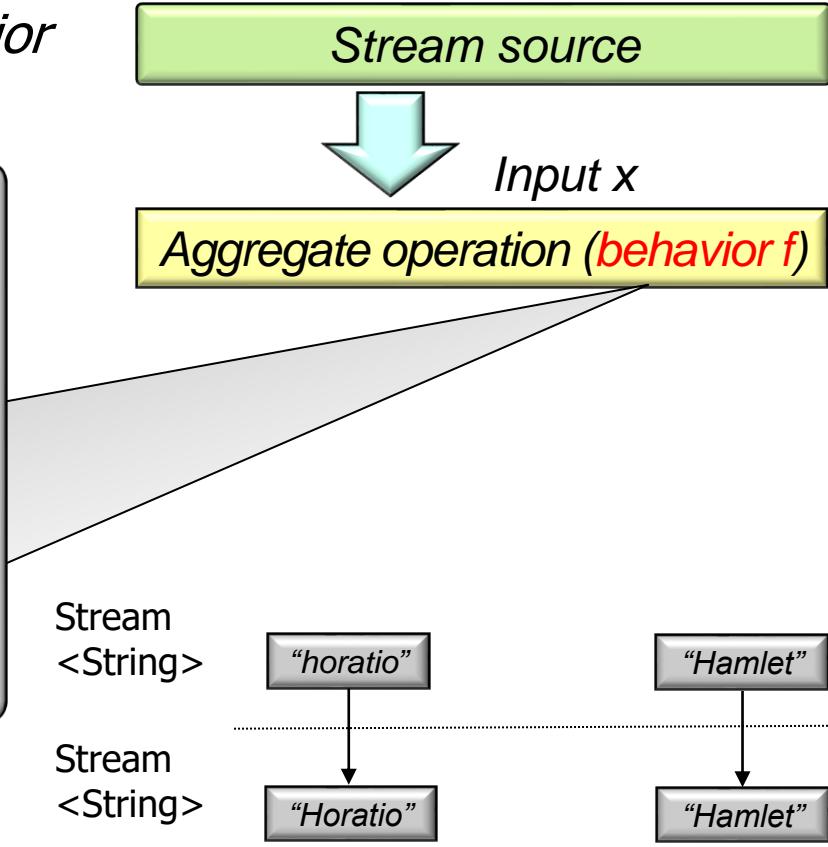
# Java Streams Aggregate Operations

- An aggregate operation performs a *behavior* on elements in a stream

**Stream**

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```

*Method reference*



# Java Streams Aggregate Operations

---

- An aggregate operation performs a *behavior* on elements in a stream
  - Some aggregate operations perform behaviors on all elements in a stream



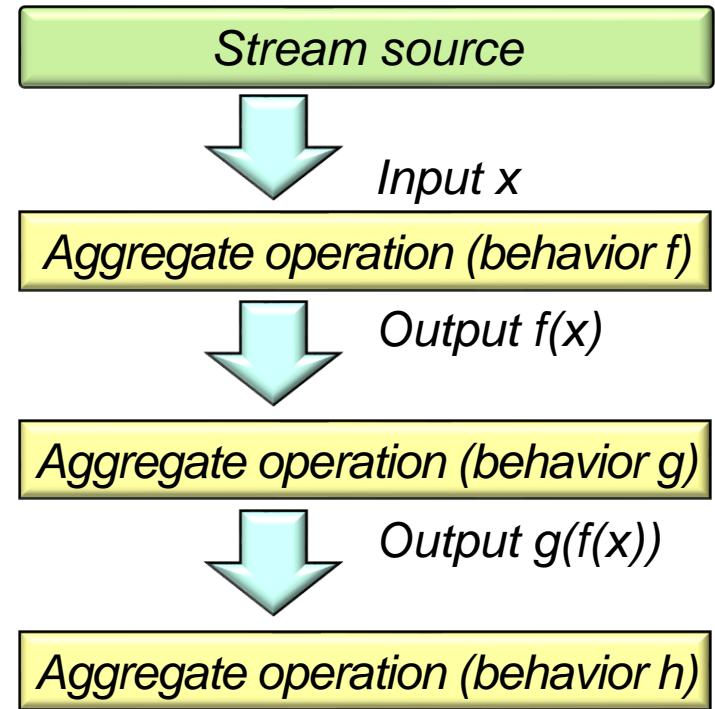
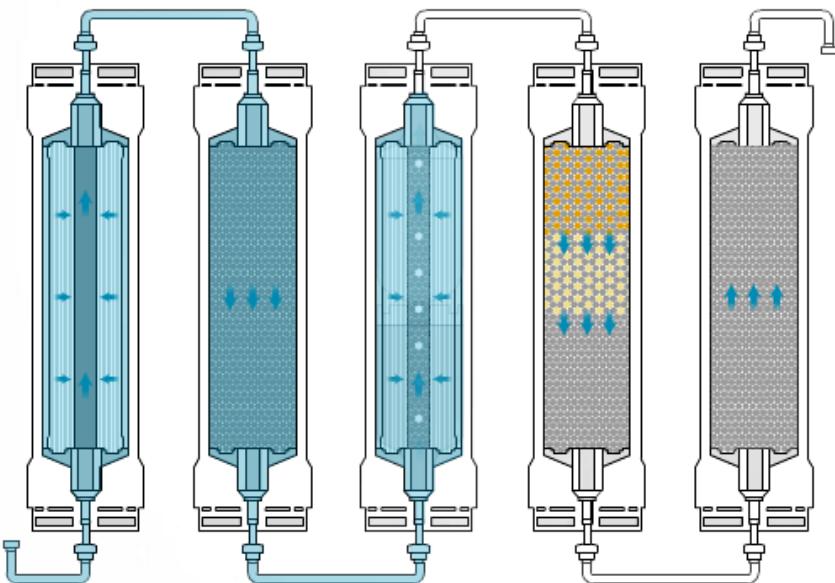
# Java Streams Aggregate Operations

- An aggregate operation performs a *behavior* on elements in a stream
  - Some aggregate operations perform behaviors on all elements in a stream
  - Other aggregate operations perform behaviors on some elements in a stream



# Java Streams Aggregate Operations

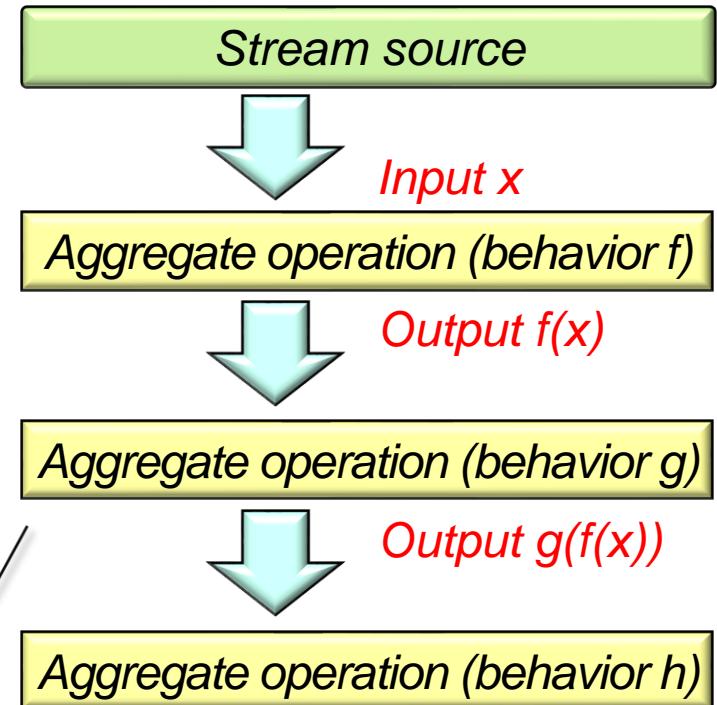
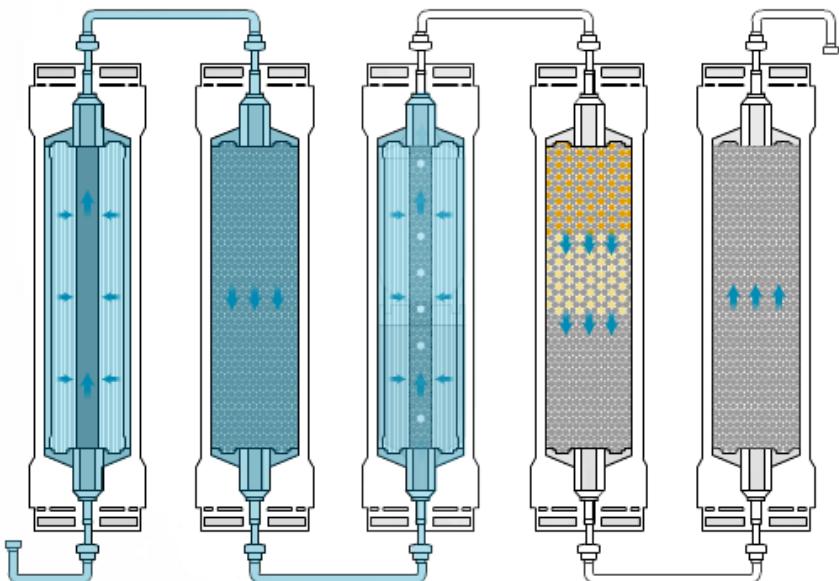
- Aggregate operations can be composed to form a pipeline of processing phases



See [en.wikipedia.org/wiki/Pipeline\\_\(software\)](https://en.wikipedia.org/wiki/Pipeline_(software))

# Java Streams Aggregate Operations

- Aggregate operations can be composed to form a pipeline of processing phases



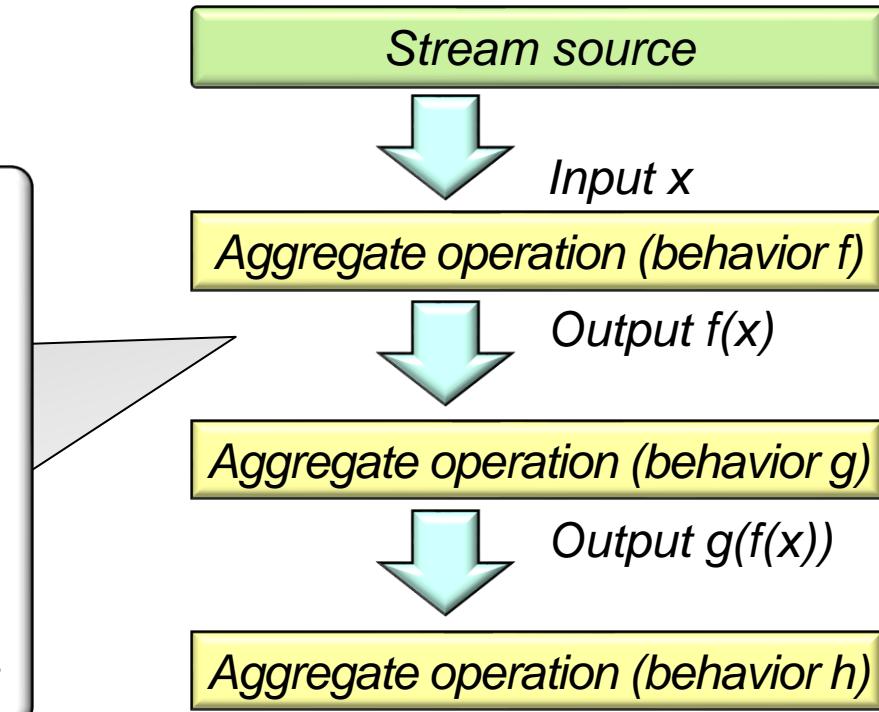
*The output of one aggregate operation can be input into the next one in the stream.*

# Java Streams Aggregate Operations

- Aggregate operations can be composed to form a pipeline of processing phases

Stream

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



*Java streams supports pipelining of aggregate operations via "fluent interfaces".*

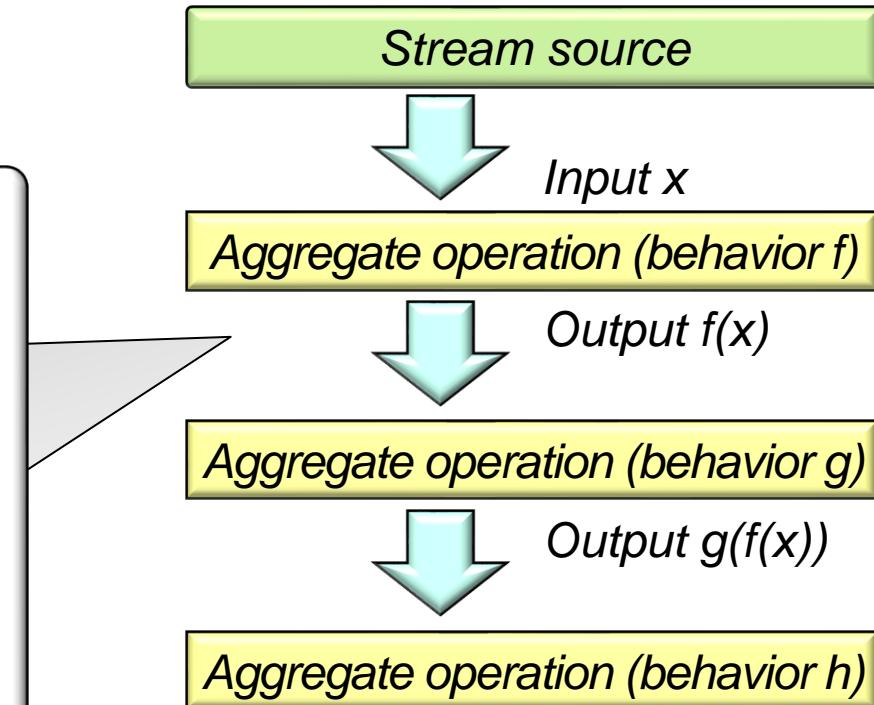
See [en.wikipedia.org/wiki/Fluent\\_interface](https://en.wikipedia.org/wiki/Fluent_interface)

# Java Streams Aggregate Operations

- Aggregate operations can be composed to form a pipeline of processing phases

**Stream**

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



*A factory method that creates a stream from an array of elements*

See upcoming lessons on “*Stream Creation Operations*”

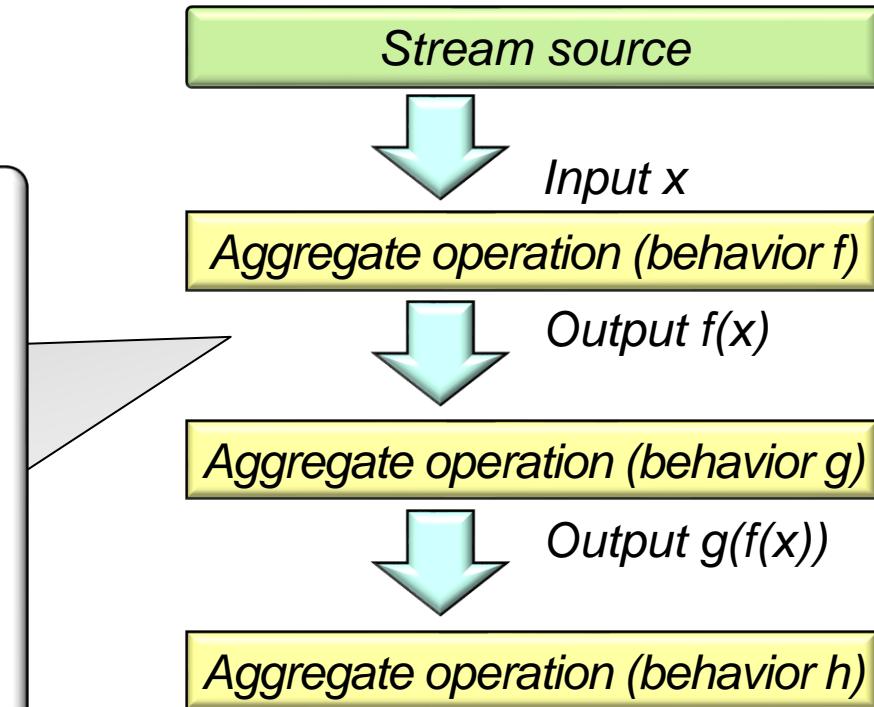
# Java Streams Aggregate Operations

- Aggregate operations can be composed to form a pipeline of processing phases

**Stream**

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```

*An aggregate operation that returns a stream containing only elements matching the predicate*



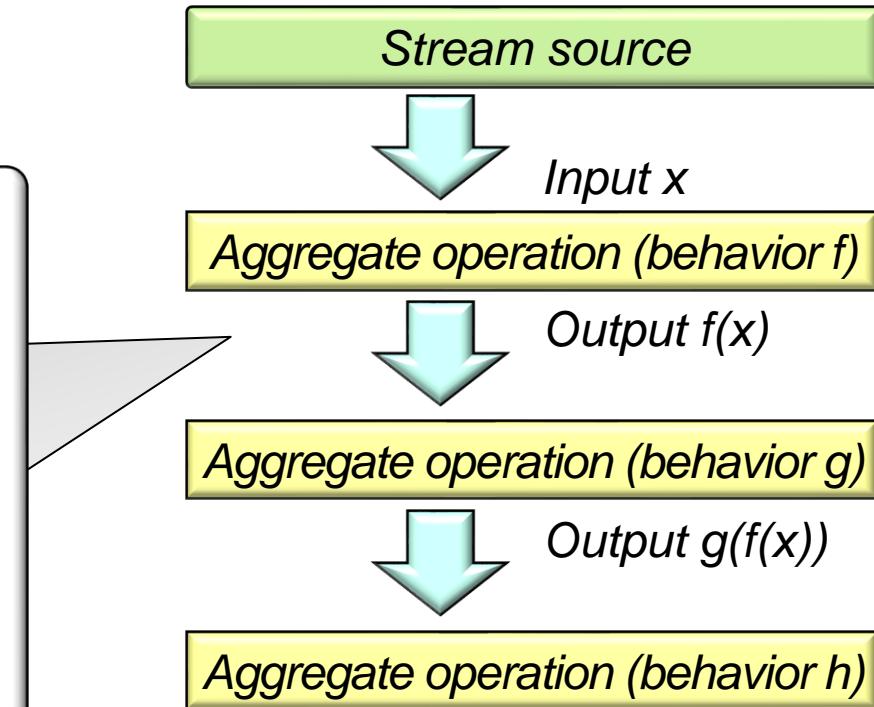
See upcoming lessons on "Stream Intermediate Operations"

# Java Streams Aggregate Operations

- Aggregate operations can be composed to form a pipeline of processing phases

**Stream**

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



*An aggregate operation that returns a stream consisting of results of applying a function to elements of this stream*

See upcoming lessons on "Stream Intermediate Operations"

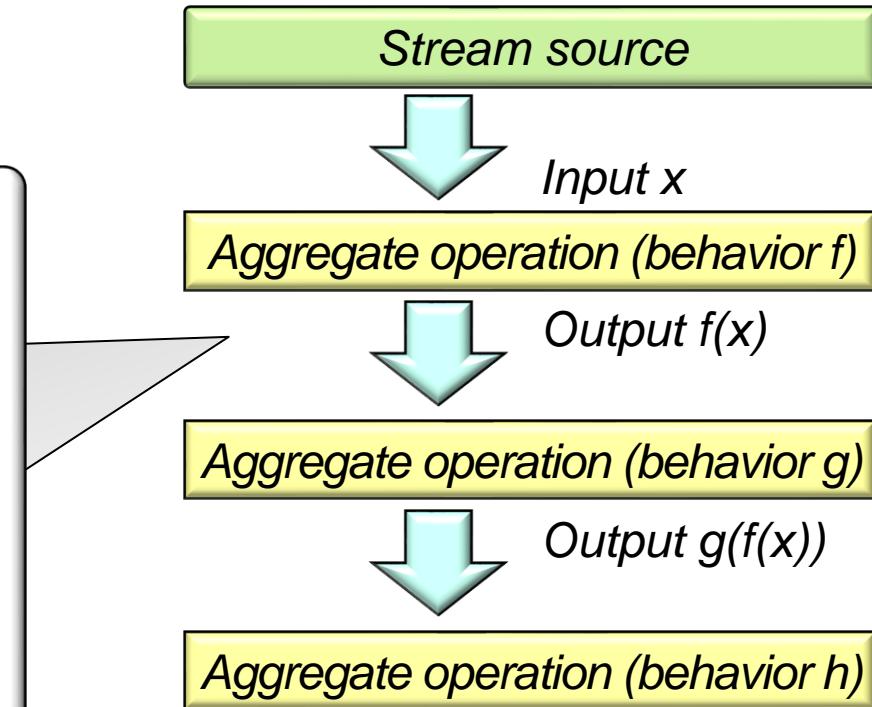
# Java Streams Aggregate Operations

- Aggregate operations can be composed to form a pipeline of processing phases

**Stream**

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```

*An aggregate operation that returns a stream consisting of results sorted in the natural order*



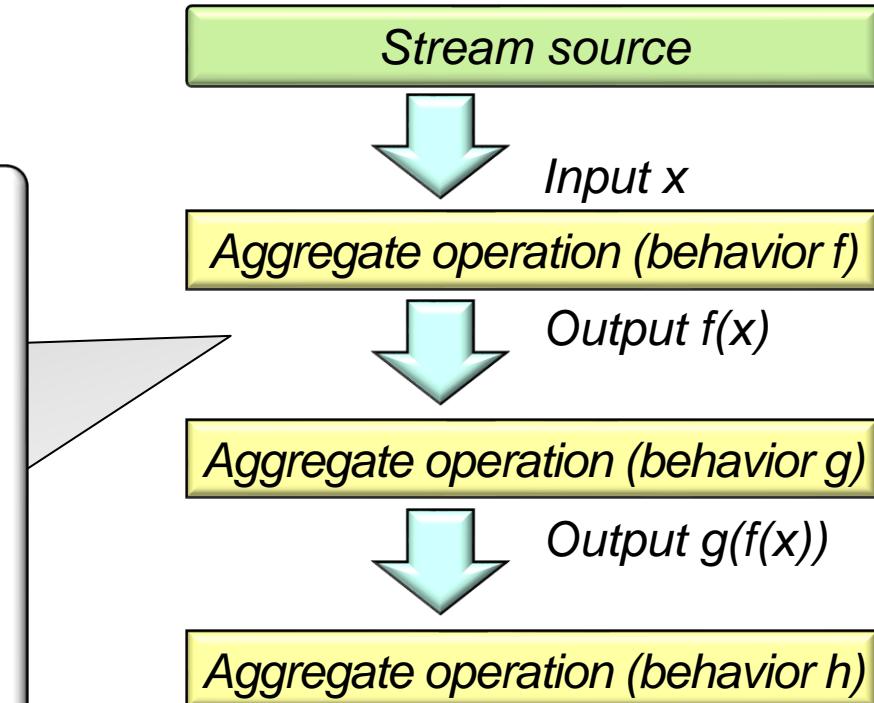
See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#sorted](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#sorted)

# Java Streams Aggregate Operations

- Aggregate operations can be composed to form a pipeline of processing phases

**Stream**

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



*An aggregate operation that performs an action on each element of the stream*

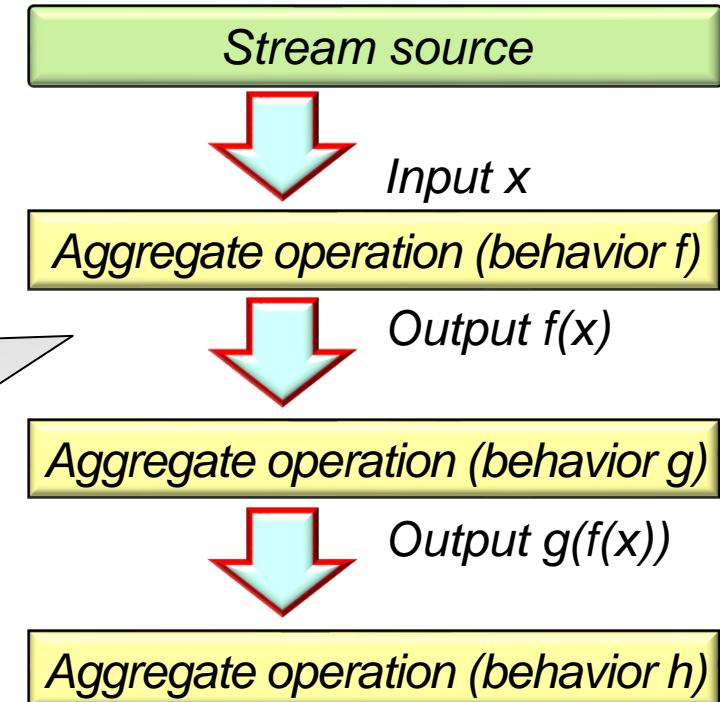
See upcoming lessons on "Stream Terminal Operations"

# Java Streams Aggregate Operations

- Java streams iterate internally (& invisibly) between aggregate operations

**Stream**

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



*Internal iteration enhances opportunities for transparent optimization & incurs fewer accidental complexities*

# Java Streams Aggregate Operations

- In contrast, collections are iterated explicitly using loops and/or iterators.

```
List<String> l = new LinkedList<>()
(List.of("horatio", "laertes", "Hamlet", ...));

for (int i = 0; i < l.size();)
 if (toLowerCase(l.get(i).charAt(0)) != 'h')
 l.remove(i);
 else {
 l.set(i, capitalize(l.get(i))); i++;
 }

Collections.sort(l);

for (String s : l) System.out.println(s);
```

*More opportunities for accidental complexities & harder to optimize*

---

# End of Understand Java Streams Common Operations

# Visualizing Java Streams in Action

Douglas C. Schmidt

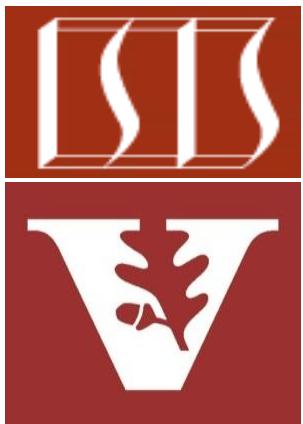
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Understand Java streams structure & functionality, e.g.
  - Fundamentals of streams
  - Three streams phases
  - Operations that create a stream
  - Aggregate operations in a stream
  - Visualizing streams in action

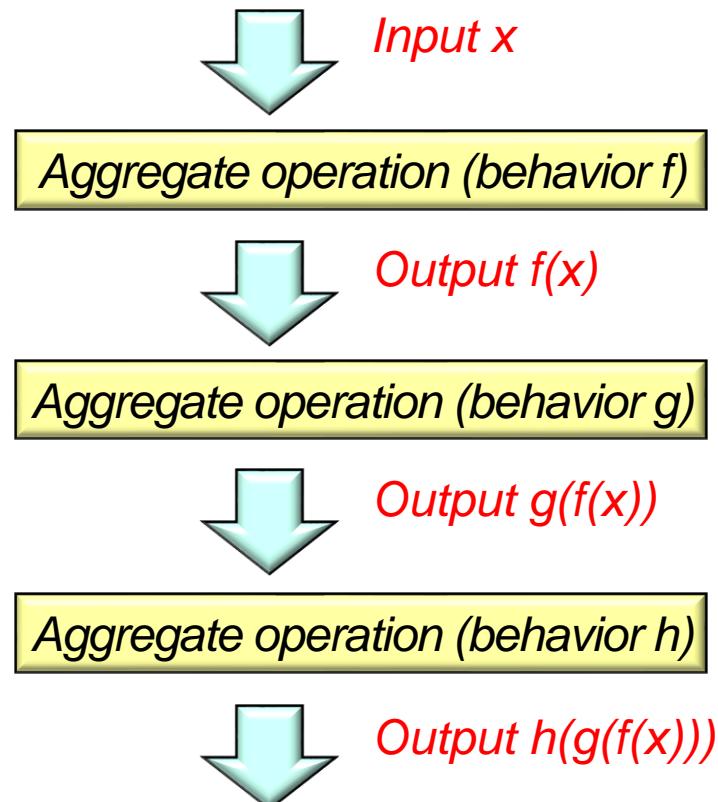


---

# Visualizing Streams in Action

# Visualizing Streams in Action

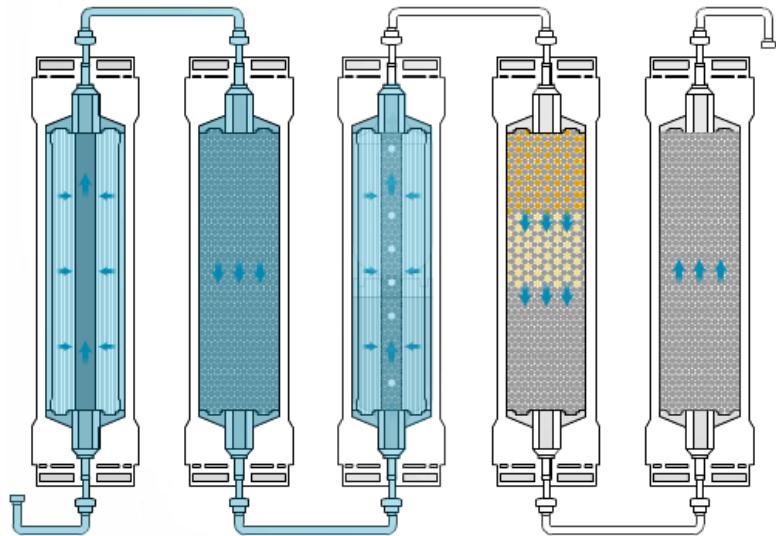
- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



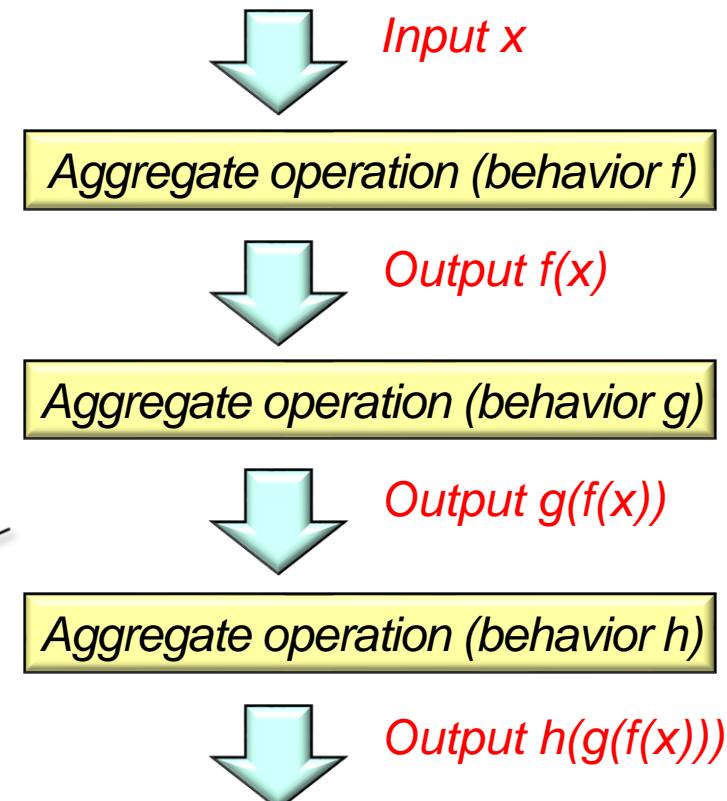
See [en.wikipedia.org/wiki/Pipeline\\_\(software\)](https://en.wikipedia.org/wiki/Pipeline_(software))

# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together

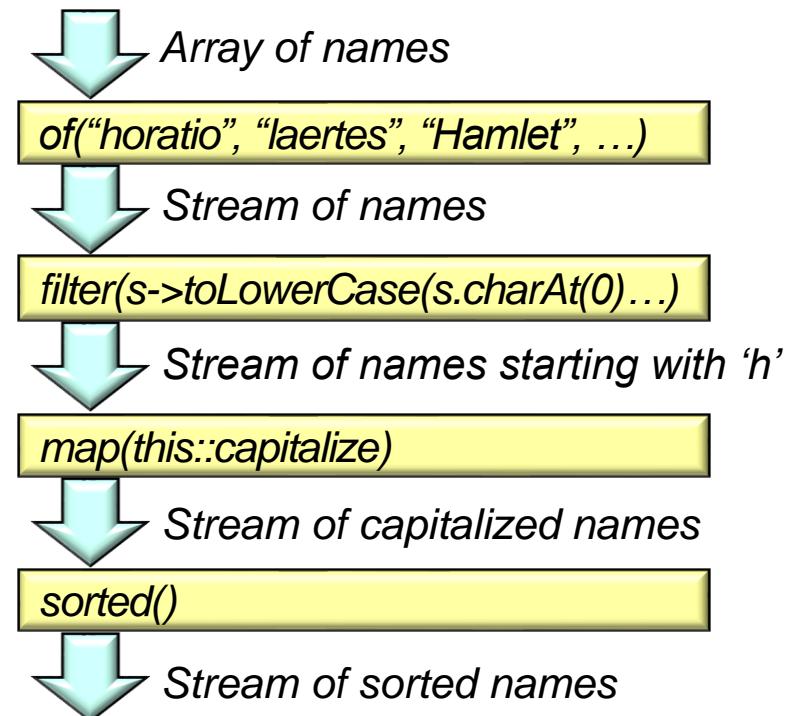
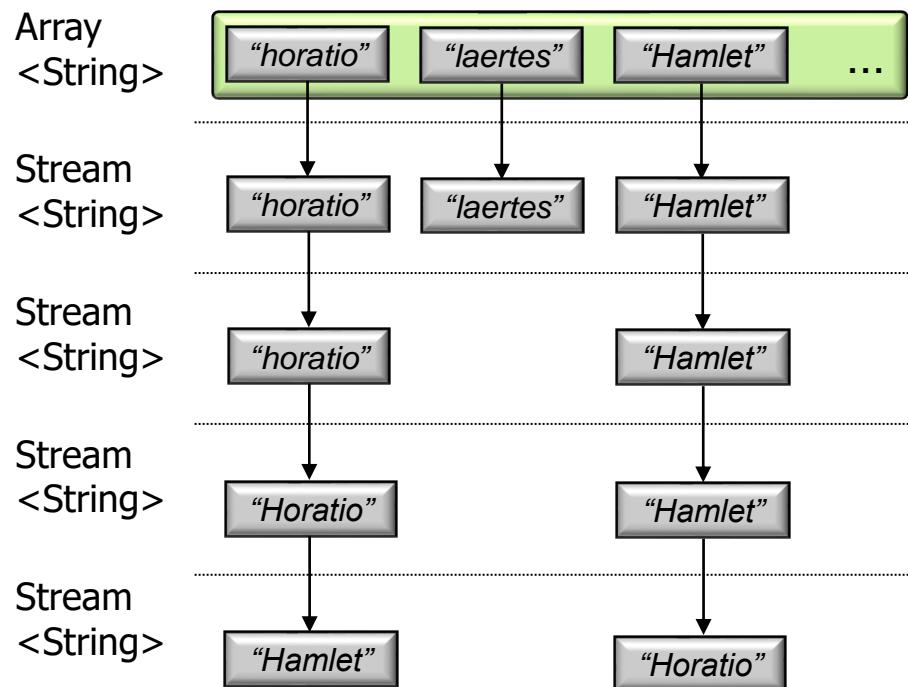


*Each aggregate operation in the pipeline can filter and/or transform the stream.*



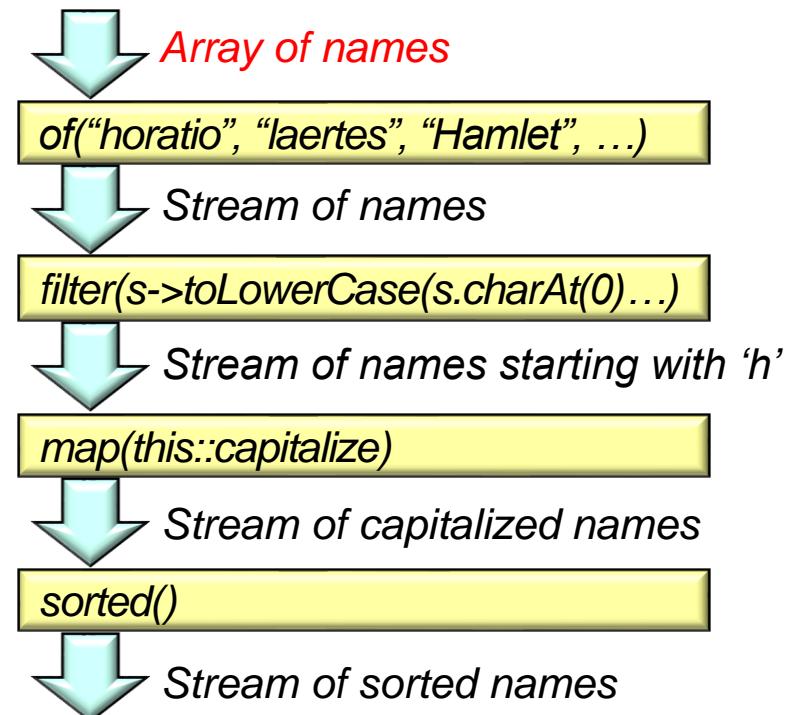
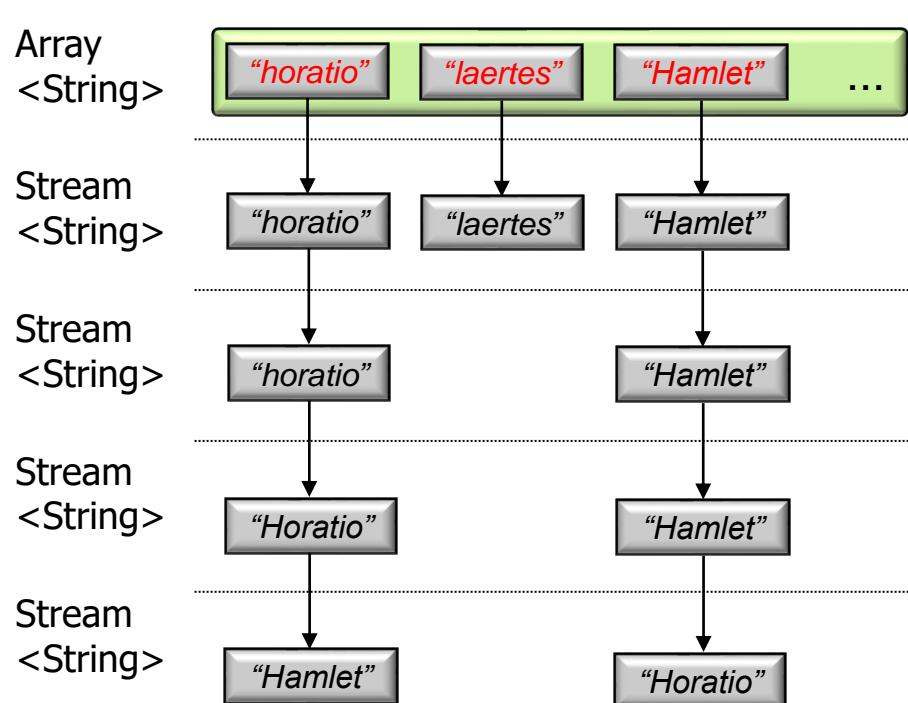
# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



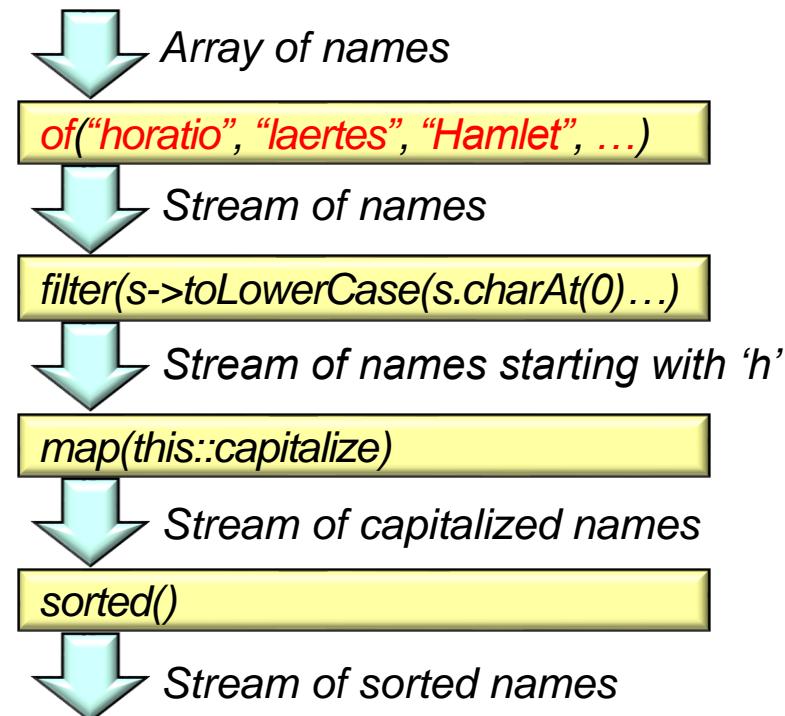
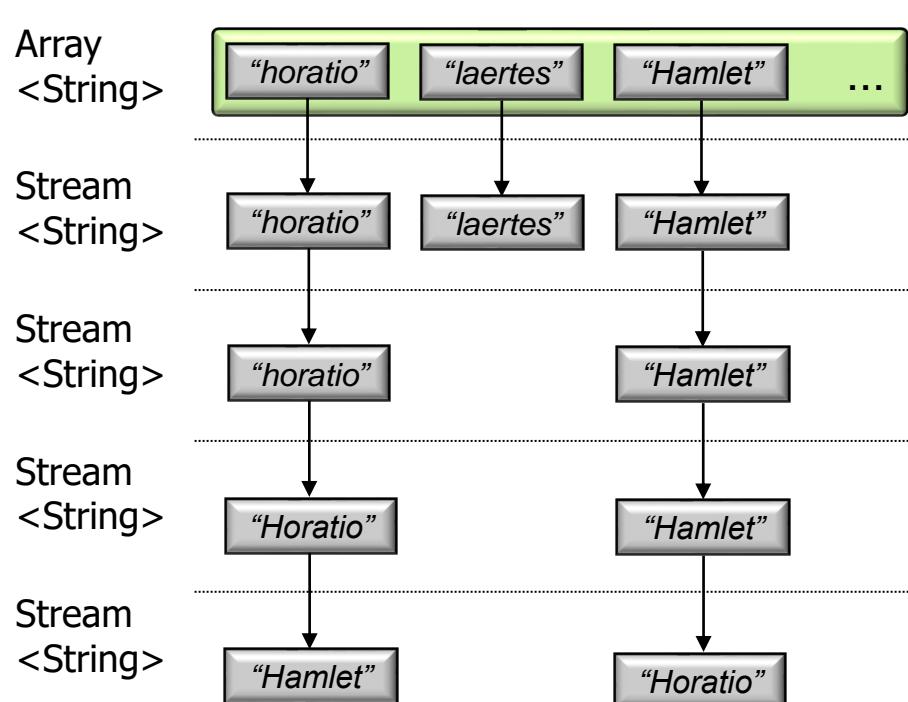
# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



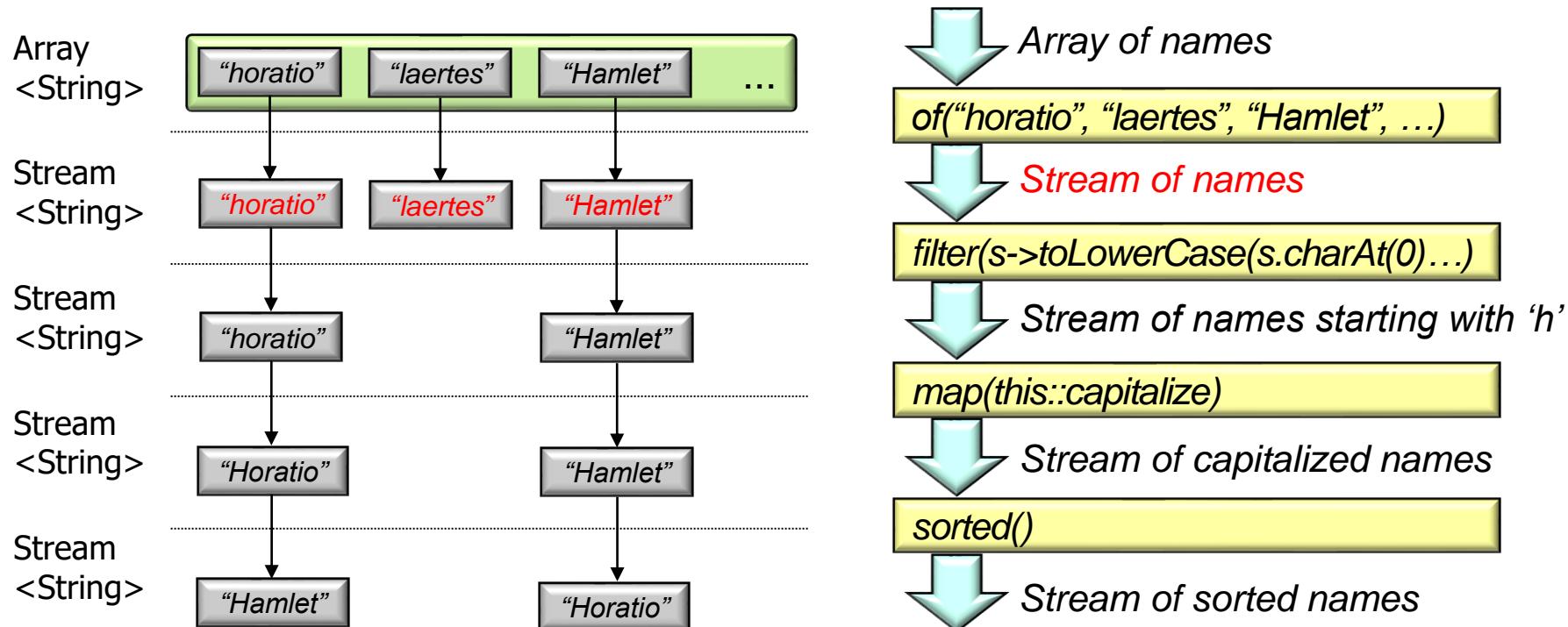
# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



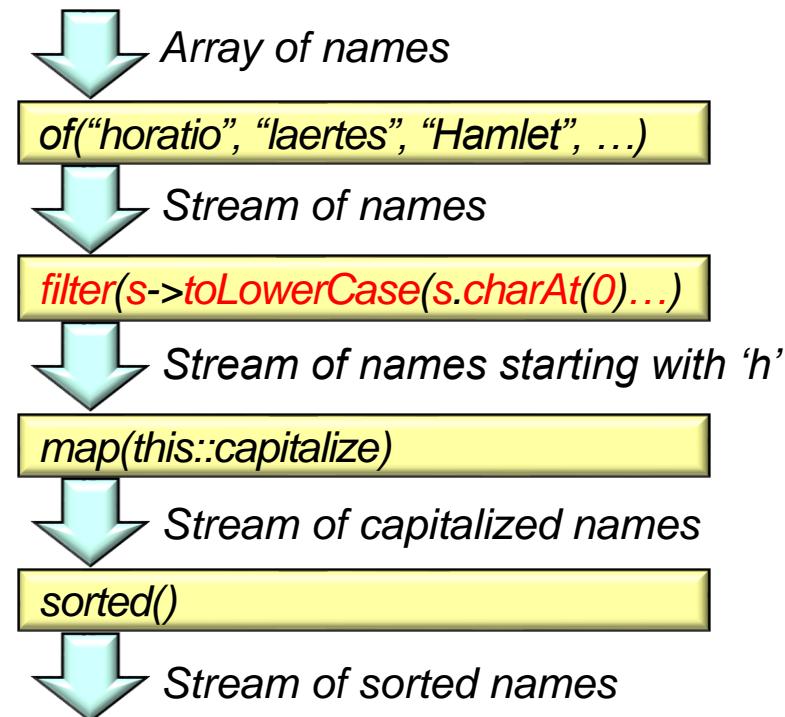
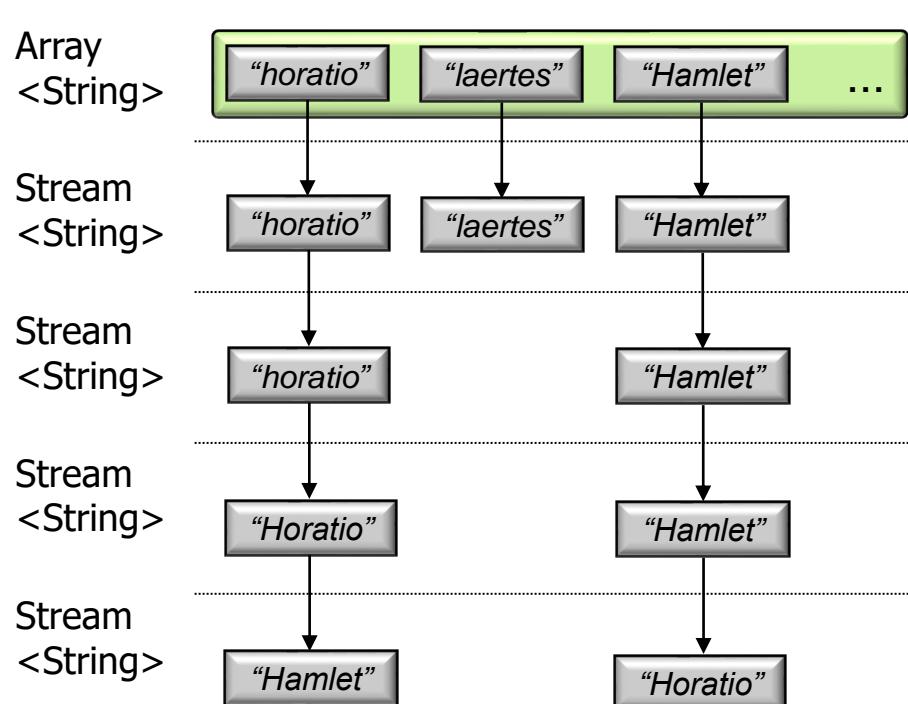
# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



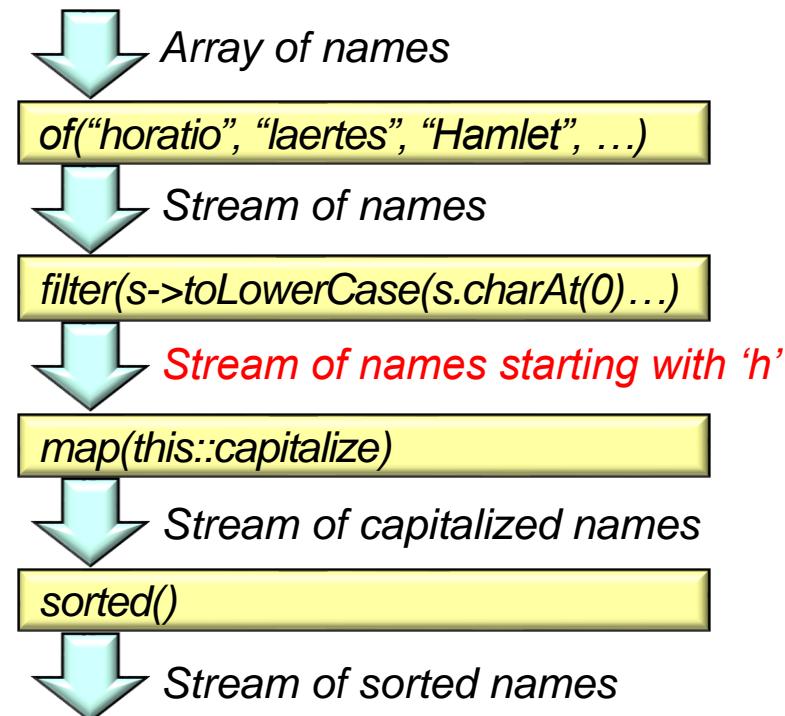
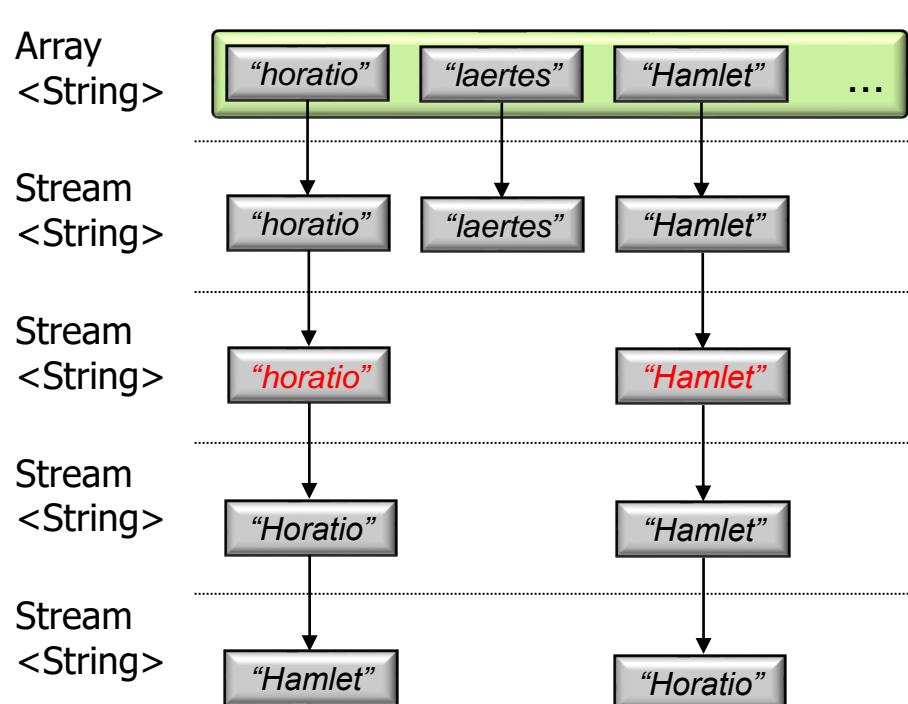
# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



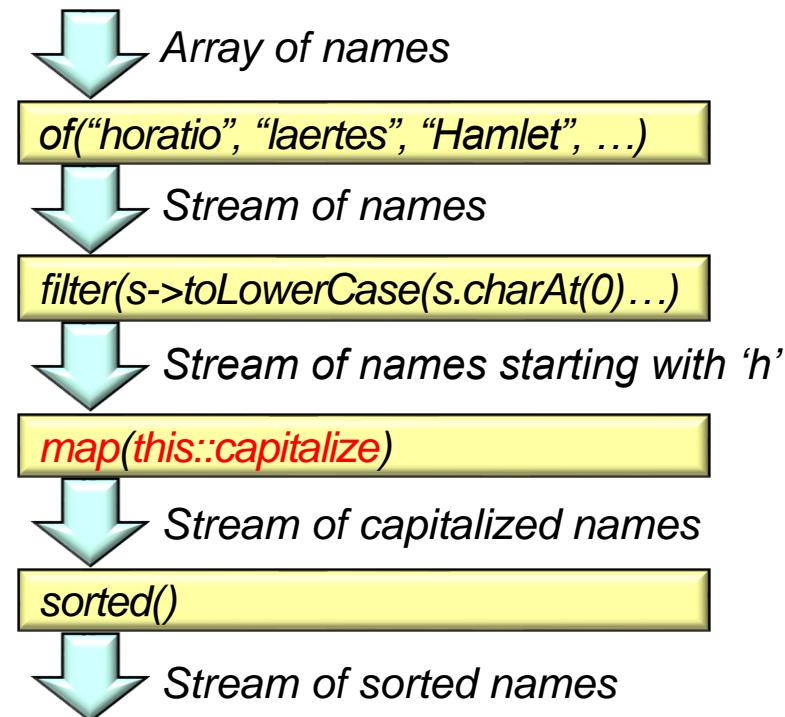
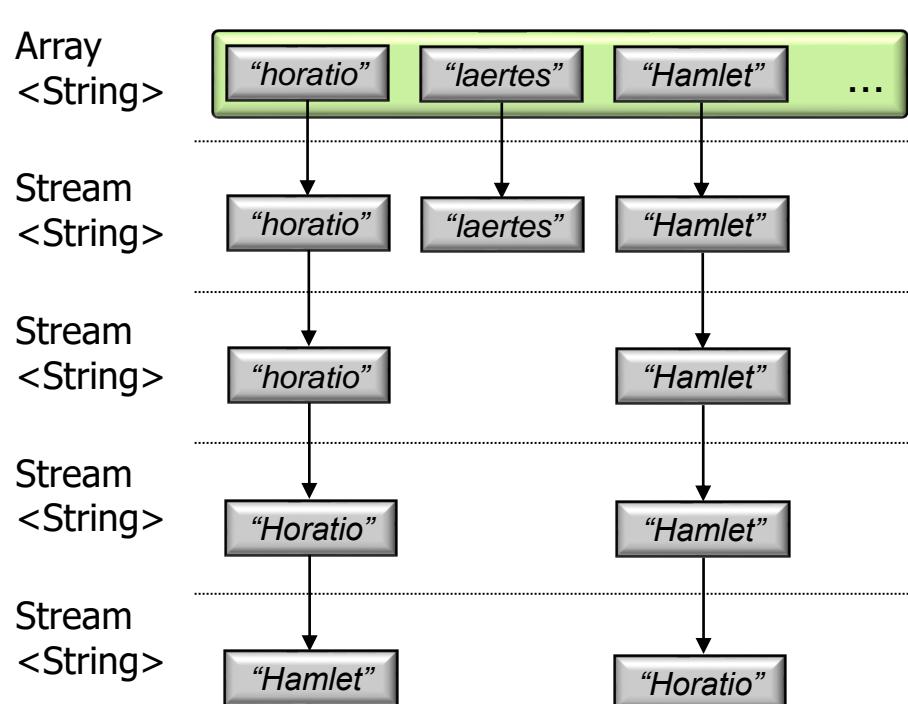
# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



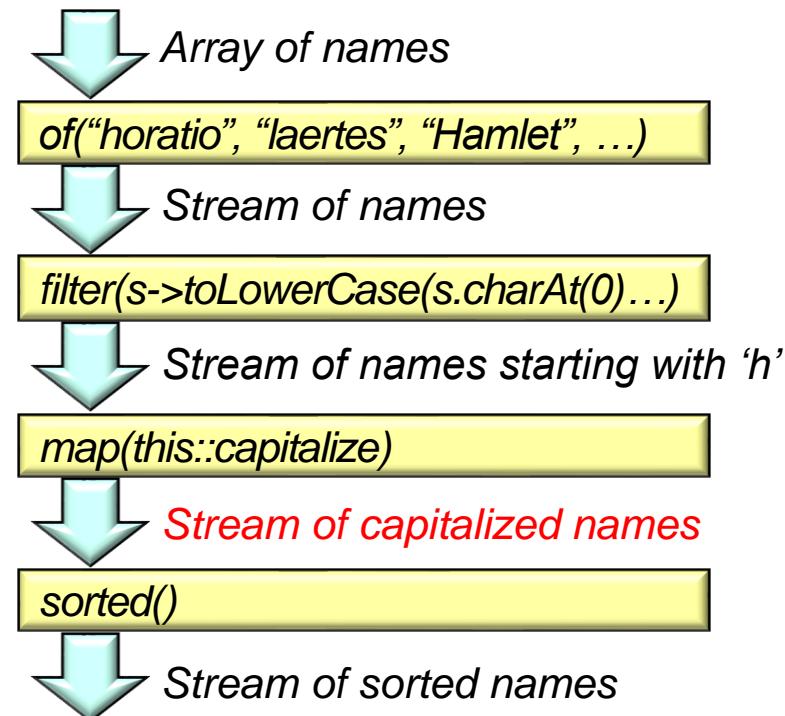
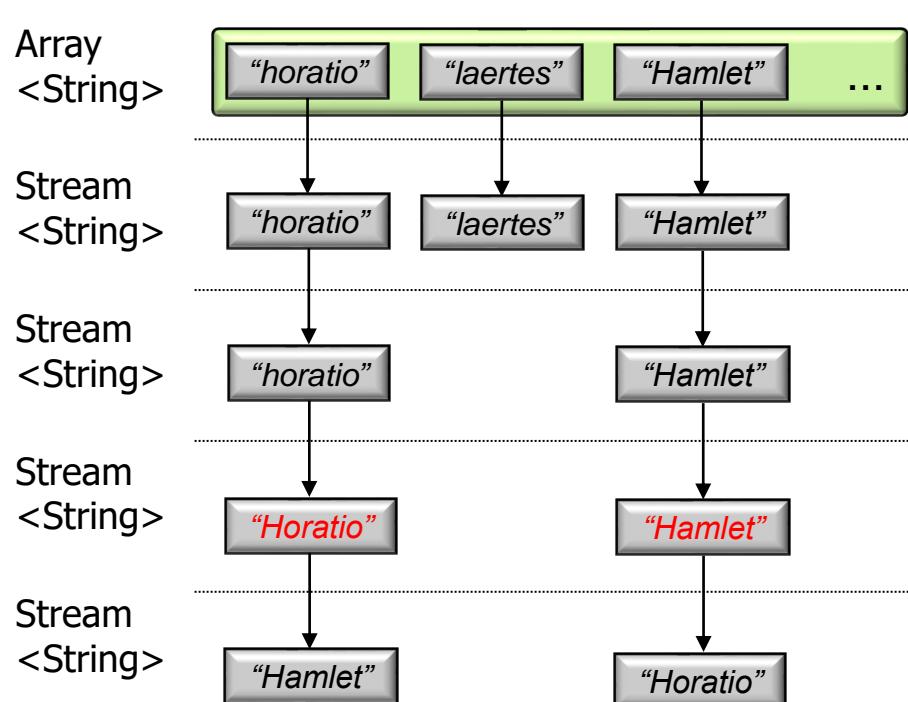
# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



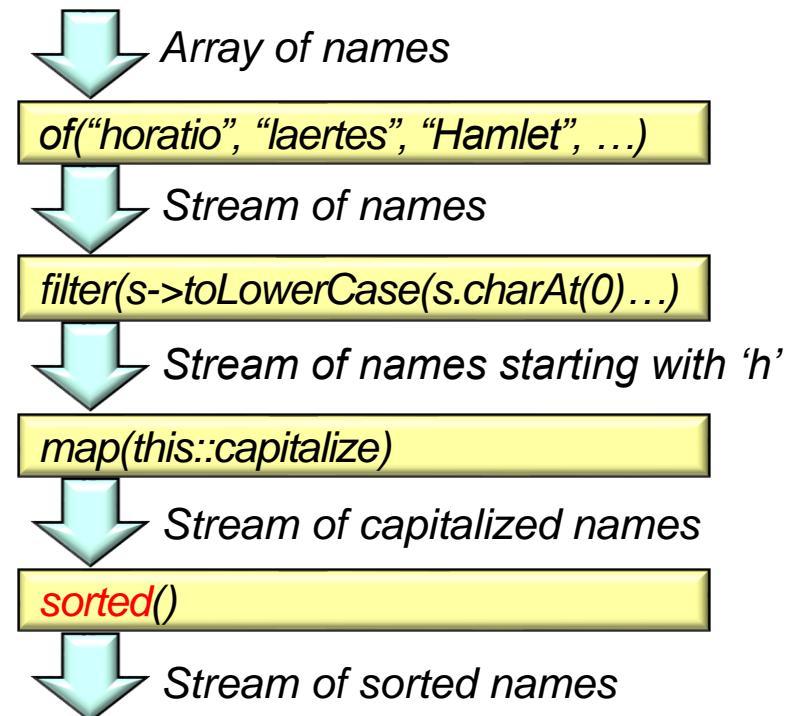
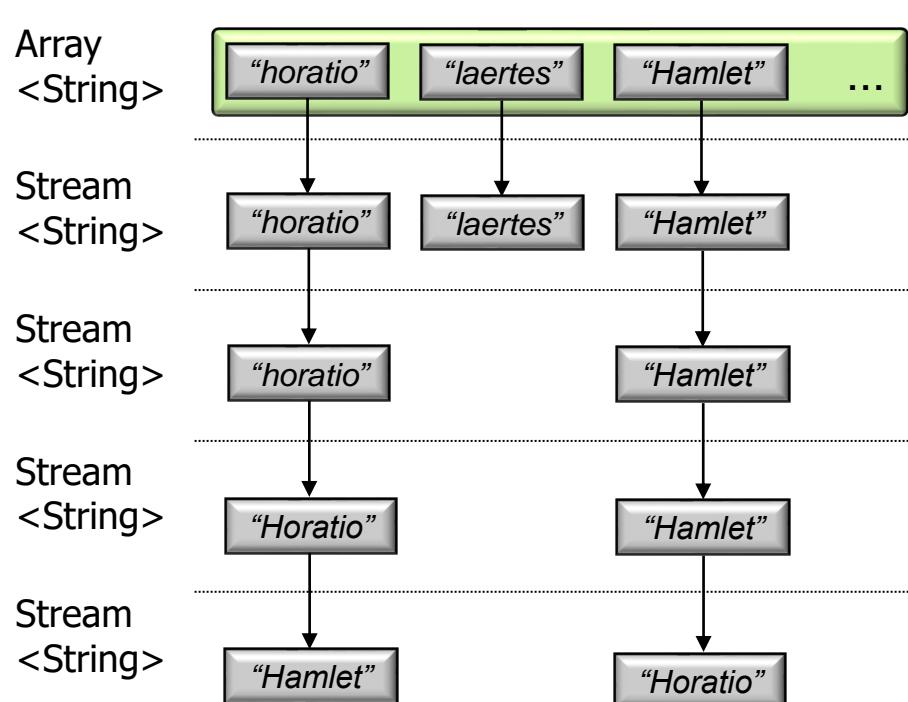
# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



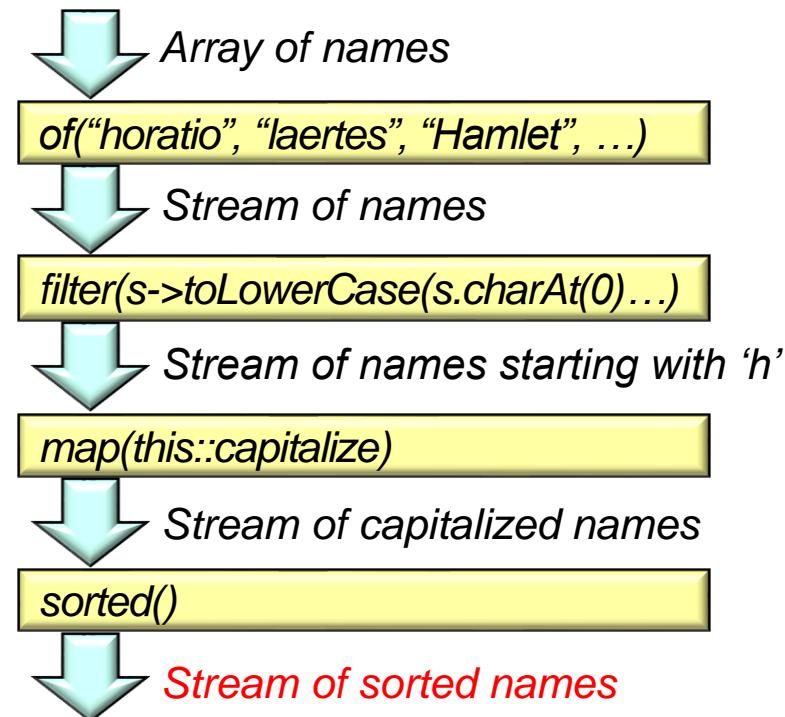
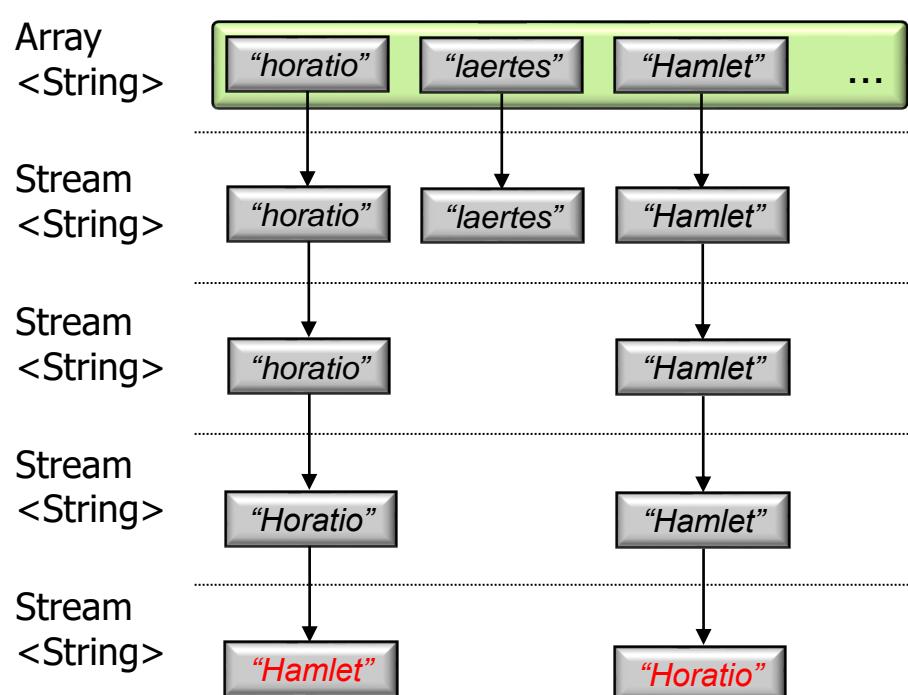
# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



# Visualizing Streams in Action

- Streams enhance flexibility by forming a “processing pipeline” that composes multiple aggregate operations together



---

# End of Visualizing Java Streams in Action

# Comparing Java Sequential Streams with Parallel Streams

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

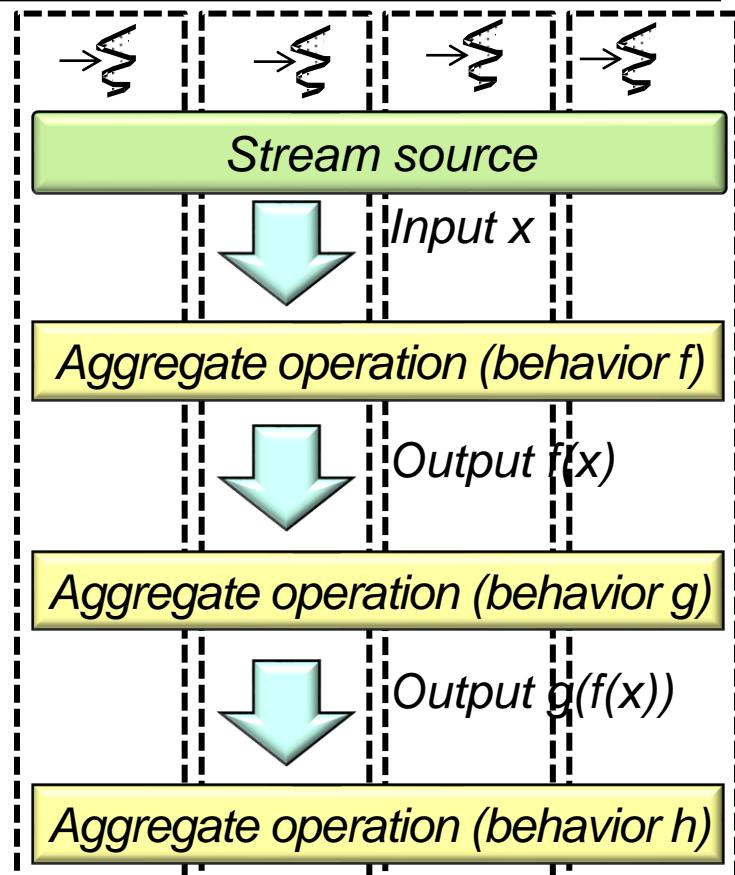
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



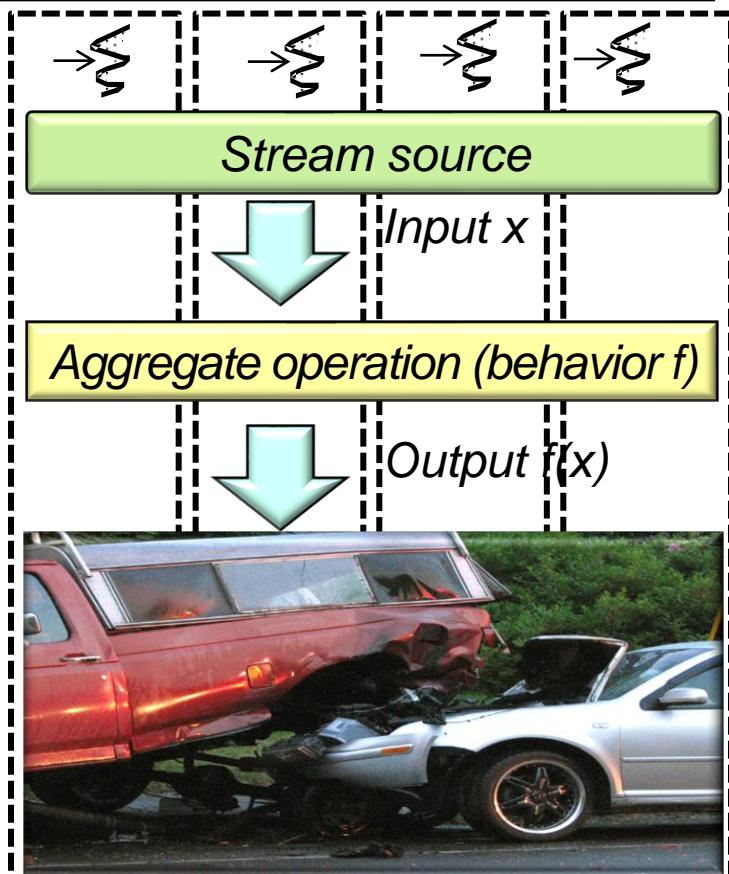
# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java streams, e.g.,
  - Fundamentals of streams
  - Benefits of streams
  - Creating a stream
  - Aggregate operations in a stream
  - Applying streams in practice
  - Sequential vs. parallel streams



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java streams, e.g.,
  - Fundamentals of streams
  - Benefits of streams
  - Creating a stream
  - Aggregate operations in a stream
  - Applying streams in practice
  - Sequential vs. parallel streams
    - Common programming hazards of parallel streams



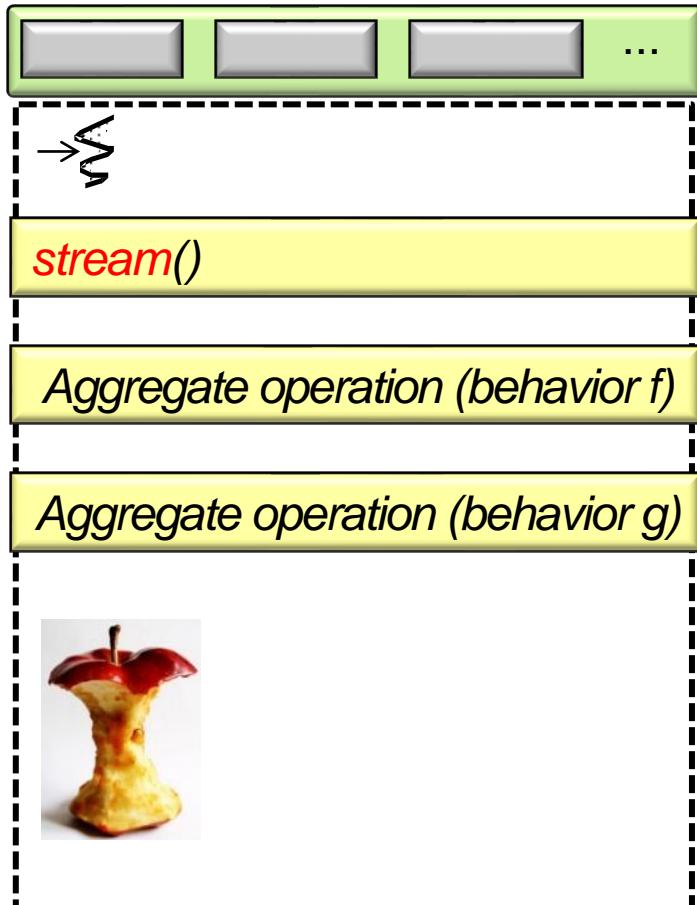
---

# Comparing Sequential vs. Parallel Streams

# Comparing Sequential vs. Parallel Streams

- Stream operations run sequentially

*We'll cover sequential streams first*

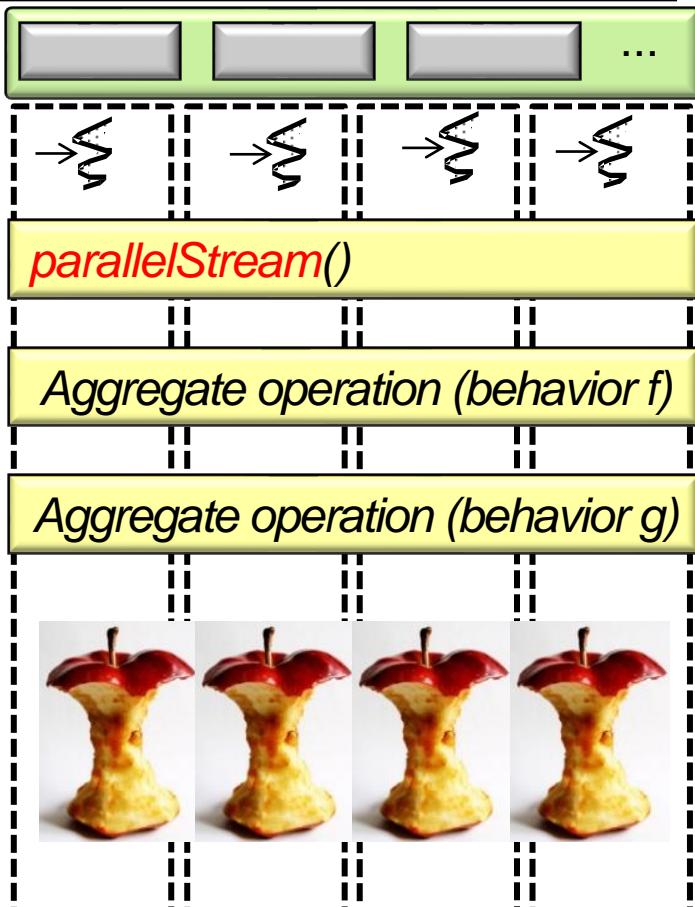


See [docs.oracle.com/javase/tutorial/collections/streams](https://docs.oracle.com/javase/tutorial/collections/streams)

# Comparing Sequential vs. Parallel Streams

- Stream operations run sequentially or in parallel

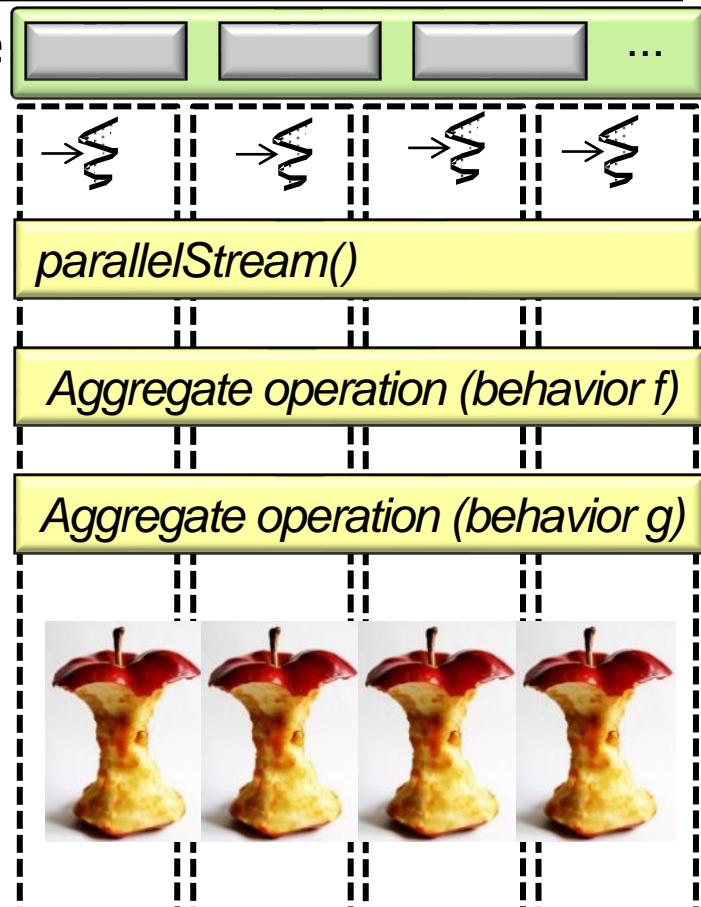
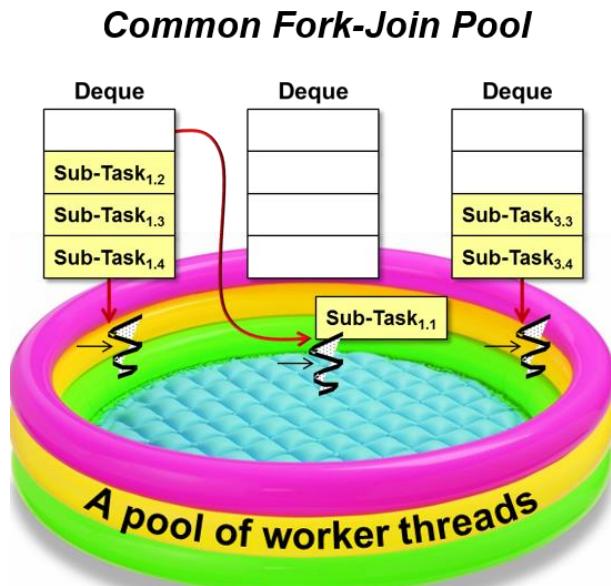
*We'll cover parallel streams later*



See [docs.oracle.com/javase/tutorial/collections/stream/parallelism.html](https://docs.oracle.com/javase/tutorial/collections/stream/parallelism.html)

# Comparing Sequential vs. Parallel Streams

- A parallel stream splits its elements into multiple chunks & uses the common fork-join pool to process these chunks independently

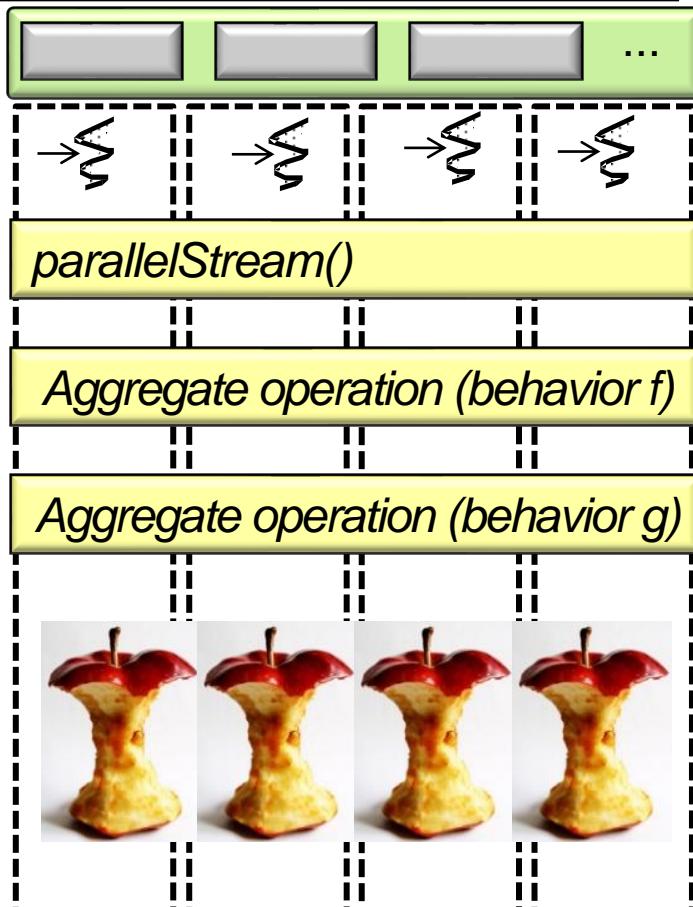
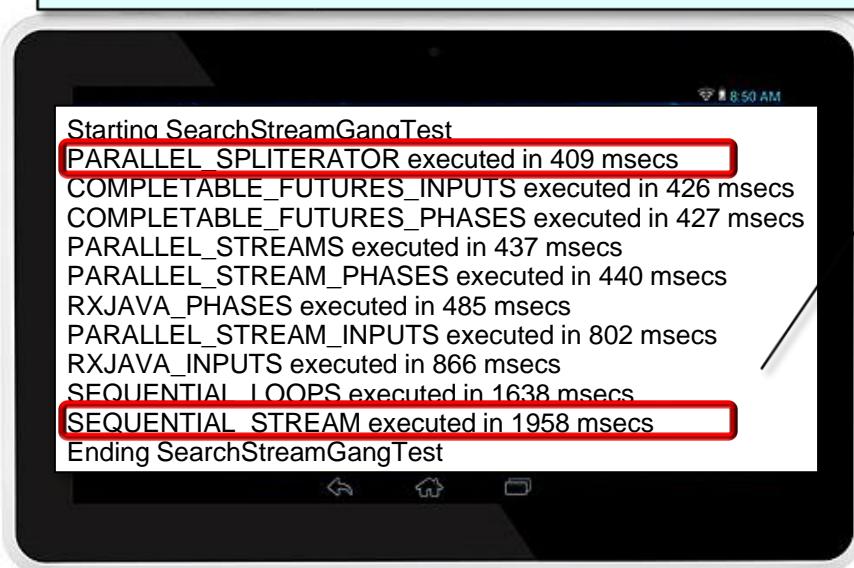


See [dzone.com/articles/common-fork-join-pool-and-streams](https://dzone.com/articles/common-fork-join-pool-and-streams)

# Comparing Sequential vs. Parallel Streams

- A parallel stream splits its elements into multiple chunks & uses the common fork-join pool to process these chunks independently

*A parallel stream can usually be much more efficient & scalable than a sequential stream.*



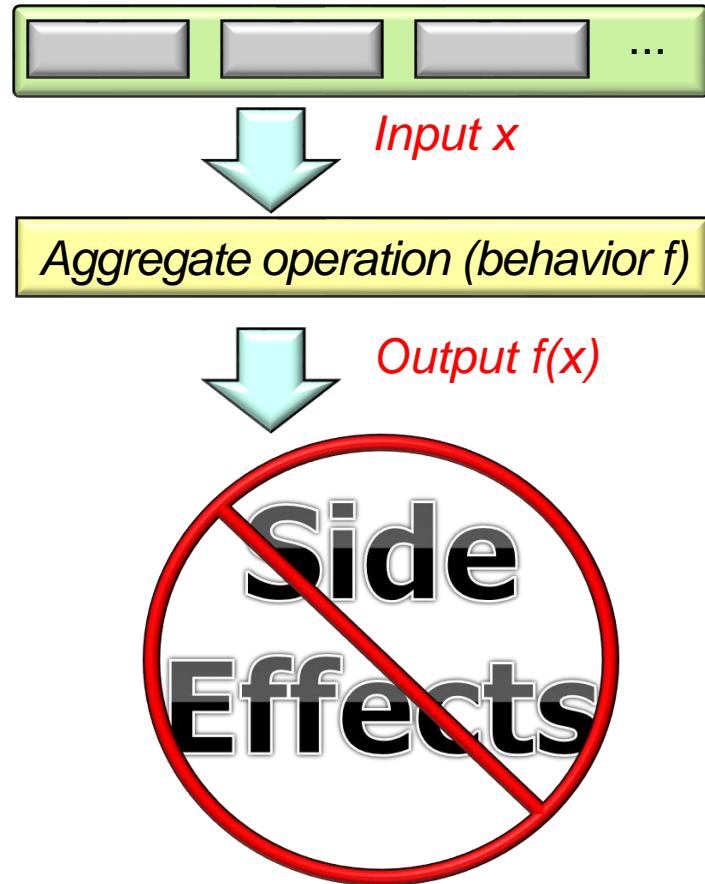
Tests conducted on a quad-core Lenovo P50 with 32 Gbytes of RAM

---

# Common Programming Hazards for Parallel Streams

# Common Programming Hazards for Parallel Streams

- Ideally, a behavior's output in a stream depends only on its input arguments

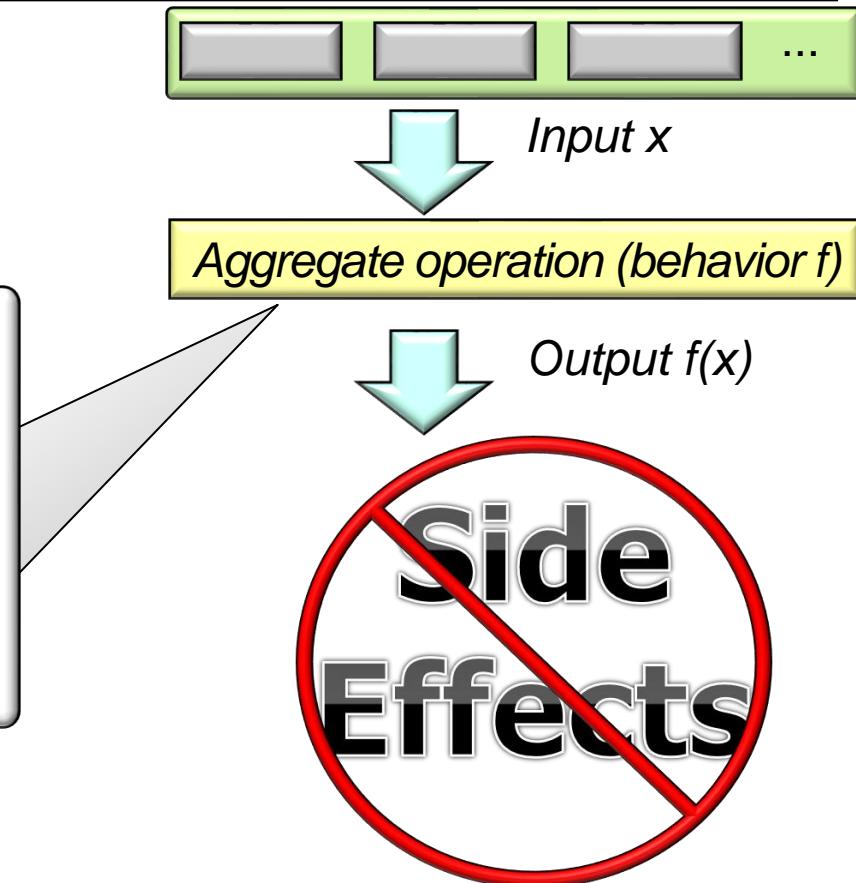


See [en.wikipedia.org/wiki/Side\\_effect\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

# Common Programming Hazards for Parallel Streams

- Ideally, a behavior's output in a stream depends only on its input arguments

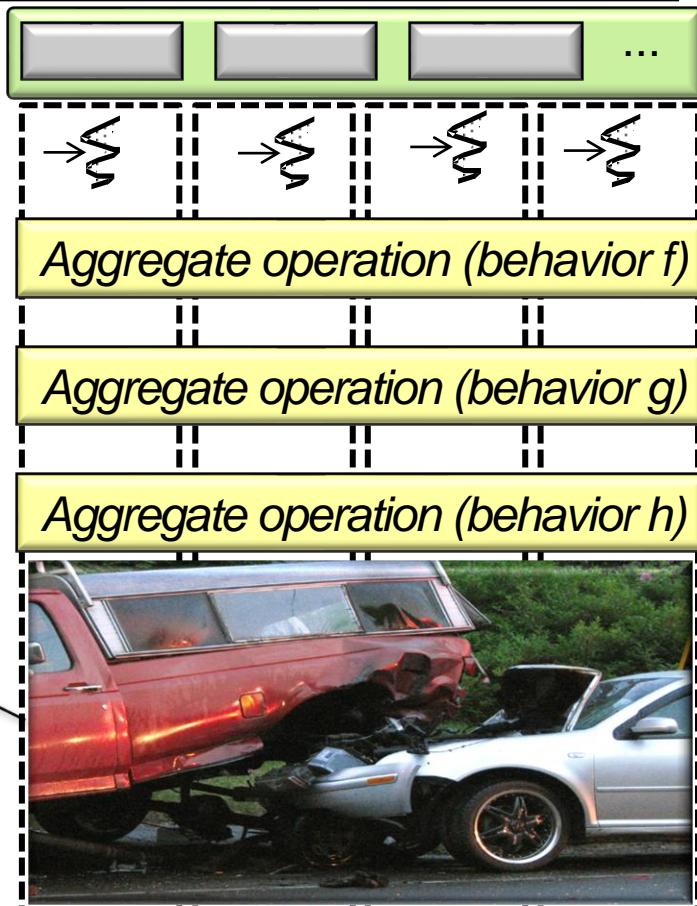
```
String capitalize(String s) {
 if (s.length() == 0)
 return s;
 return s.substring(0, 1)
 .toUpperCase()
 + s.substring(1)
 .toLowerCase();
```



# Common Programming Hazards for Parallel Streams

- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams

*Race conditions arise in software when an application depends on the sequence or timing of threads for it to operate properly*



See [en.wikipedia.org/wiki/Race\\_condition#Software](https://en.wikipedia.org/wiki/Race_condition#Software)

# Common Programming Hazards for Parallel Streams

- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```
long factorial(long n) {
 Total t = new Total();
 LongStream
 .rangeClosed(1, n)
 .parallel()
 .forEach(t::mult);
 return t.mTotal;
}
```

```
class Total {
 public long mTotal = 1;

 public void mult(long n)
 { mTotal *= n; }
```

*A buggy attempt to compute the 'n<sup>th</sup>' factorial in parallel*

# Common Programming Hazards for Parallel Streams

- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```
long factorial(long n) {
 Total t = new Total();
 LongStream
 .rangeClosed(1, n)
 .parallel()
 .forEach(t::mult);
 return t.mTotal;
}
```

```
class Total {
 public long mTotal = 1;

 public void mult(long n)
 { mTotal *= n; }
```

*Shared mutable state*



# Common Programming Hazards for Parallel Streams

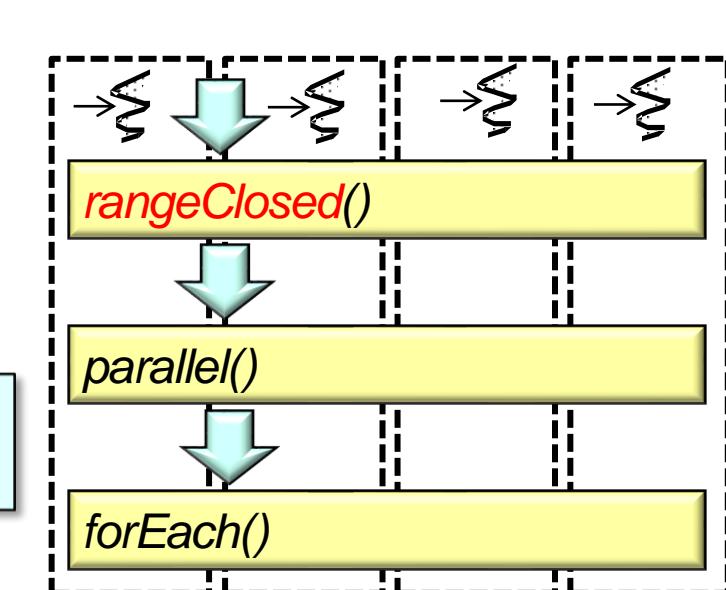
- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```
long factorial(long n) {
 Total t = new Total();
 LongStream
 .rangeClosed(1, n)
 .parallel()
 .forEach(t::mult);
 return t.mTotal;
}
```

*Generate a range  
of values from 1..n*

```
class Total {
 public long mTotal = 1;

 public void mult(long n)
 { mTotal *= n; }
```



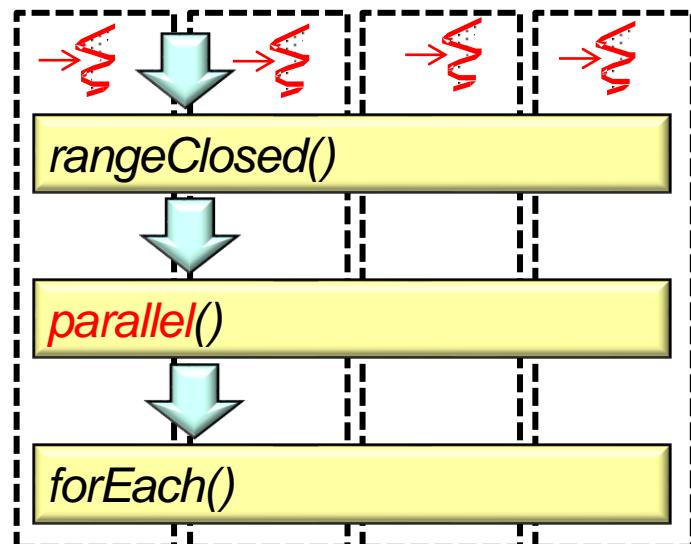
# Common Programming Hazards for Parallel Streams

- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```
long factorial(long n) {
 Total t = new Total();
 LongStream
 .rangeClosed(1, n)
 .parallel() ——————> Run in parallel
 .forEach(t::mult);
 return t.mTotal;
}
```

```
class Total {
 public long mTotal = 1;

 public void mult(long n)
 { mTotal *= n; }
```



# Common Programming Hazards for Parallel Streams

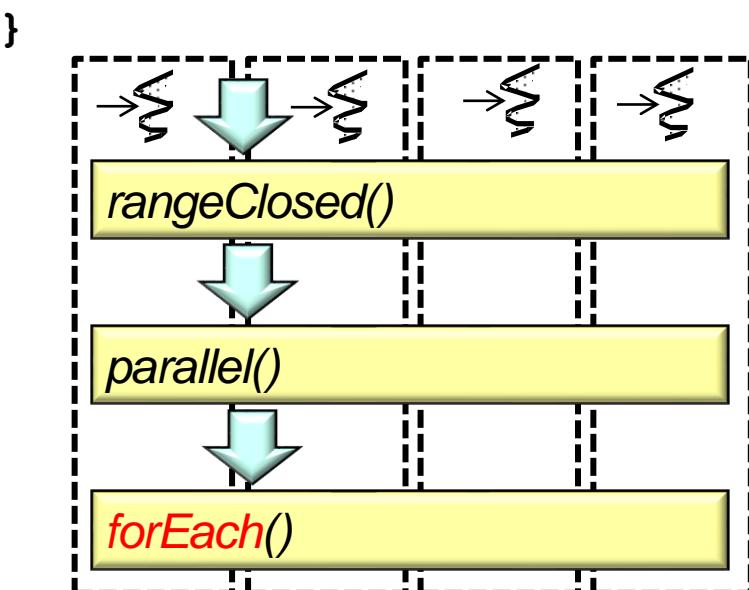
- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```
long factorial(long n) {
 Total t = new Total();
 LongStream
 .rangeClosed(1, n)
 .parallel()
 .forEach(t::mult);
 return t.mTotal;
}
```

*Multiply the running total w/the latest value*

```
class Total {
 public long mTotal = 1;

 public void mult(long n)
 { mTotal *= n; }
```



# Common Programming Hazards for Parallel Streams

- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```
long factorial(long n) {
 Total t = new Total();
 LongStream
 .rangeClosed(1, n)
 .parallel()
 .forEach(t::mult);

 return t.mTotal;
}
```

```
class Total {
 public long mTotal = 1;

 public void mult(long n)
 { mTotal *= n; }
}
```

*Beware of race conditions!!!*



See [en.wikipedia.org/wiki/Race\\_condition#Software](https://en.wikipedia.org/wiki/Race_condition#Software)

# Common Programming Hazards for Parallel Streams

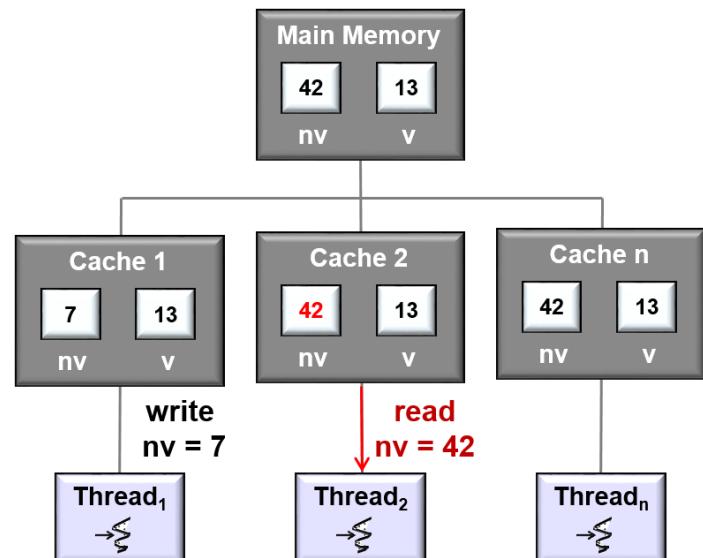
- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```
long factorial(long n) {
 Total t = new Total();
 LongStream
 .rangeClosed(1, n)
 .parallel()
 .forEach(t::mult);
 return t.mTotal;
}
```

*Beware of inconsistent memory visibility*

```
class Total {
 public long mTotal = 1;

 public void mult(long n)
 { mTotal *= n; }
```



# Common Programming Hazards for Parallel Streams

- Ideally, a behavior's output in a stream depends only on its input arguments
  - Behaviors with side-effects can incur race conditions in parallel streams, e.g.

```
long factorial(long n) {
 Total t = new Total();
 LongStream
 .rangeClosed(1, n)
 .parallel()
 .forEach(t::mult);
 return t.mTotal;
}
```

```
class Total {
 public long mTotal = 1;

 public void mult(long n)
 { mTotal *= n; }
```



***Only you can prevent  
concurrency hazards!***

In Java *you* must avoid these hazards, i.e., the compiler & JVM won't save you..

---

# End of Comparing Java Sequential Streams with Parallel Streams

# Recognize Java Streams Benefits

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

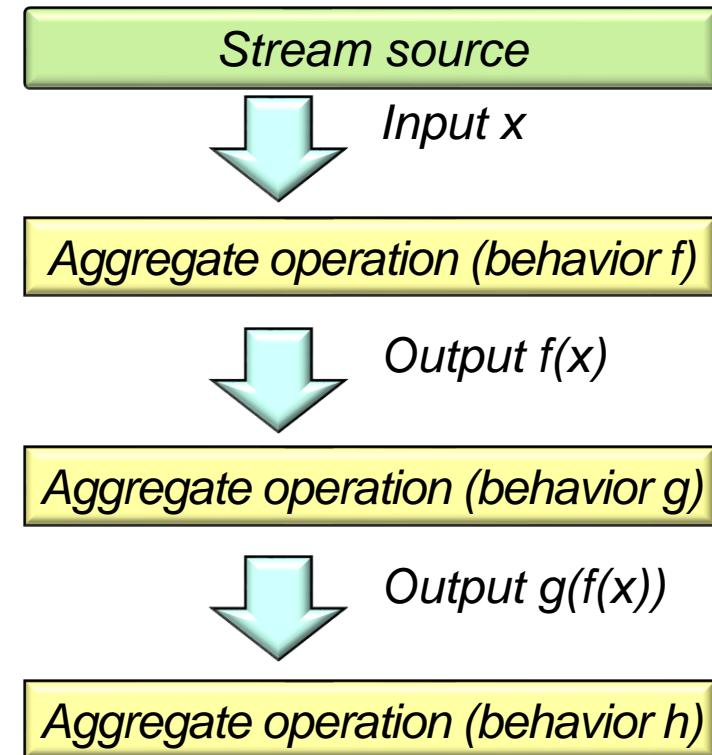
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java streams, e.g.,
  - Fundamentals of streams
  - Benefits of streams
  - Creating a stream
  - Aggregate operations in a stream
  - Applying streams in practice
  - Sequential vs. parallel streams
  - Benefits of streams



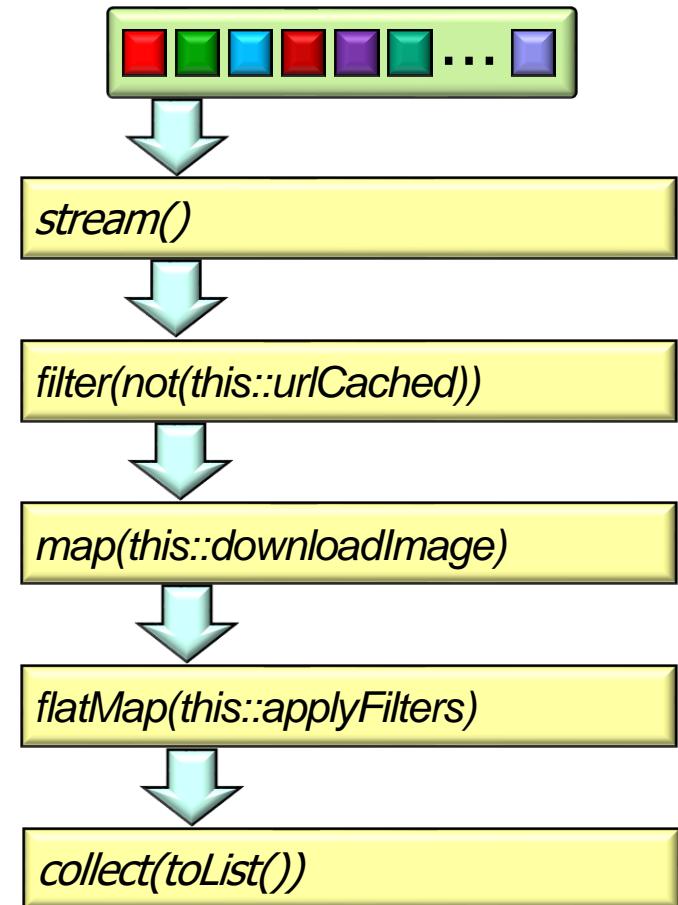
**BENEFITS**

---

# Benefits of Java Streams

# Benefits of Java Streams

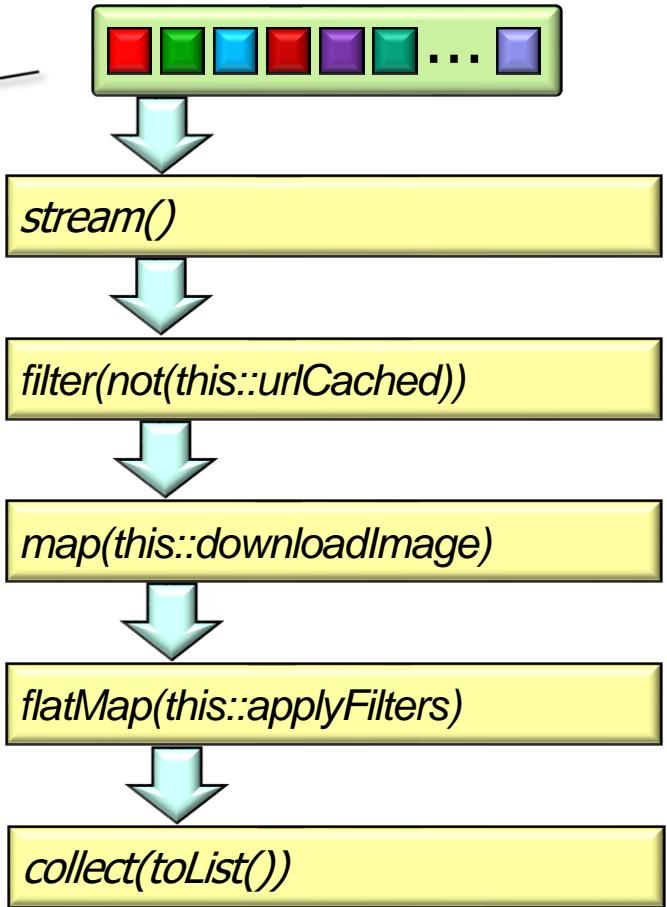
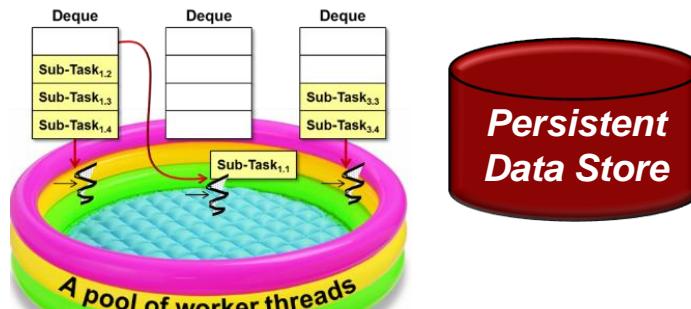
- Java streams provide several key benefits to programs & programmers



# Benefits of Java Streams

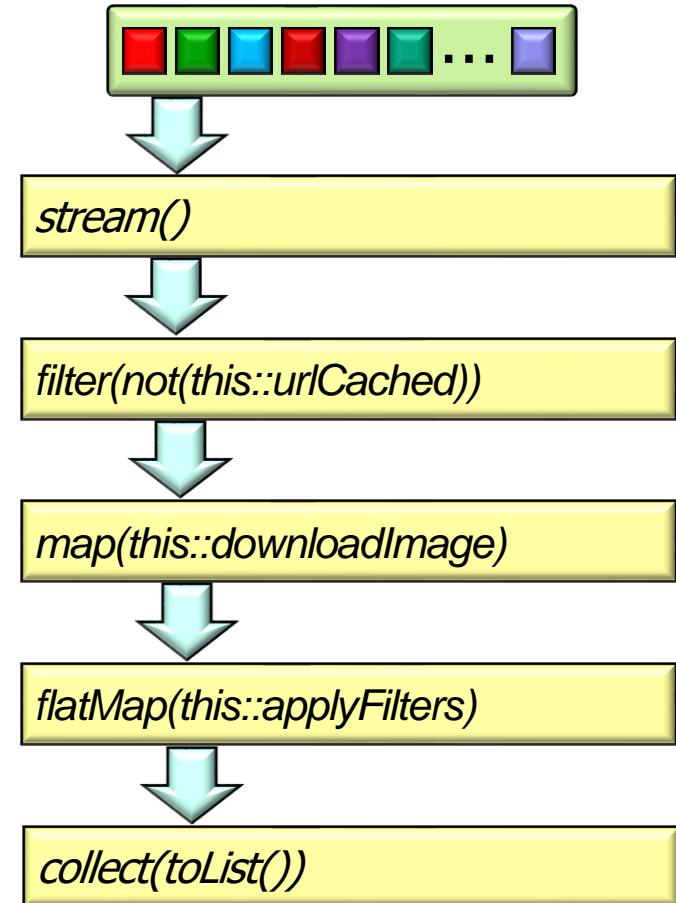
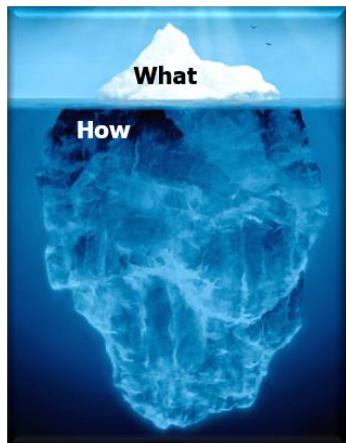
- Java streams provide several key benefits to programs & programmers

*This case study program downloads, transforms, stores, & displays images*



# Benefits of Java Streams

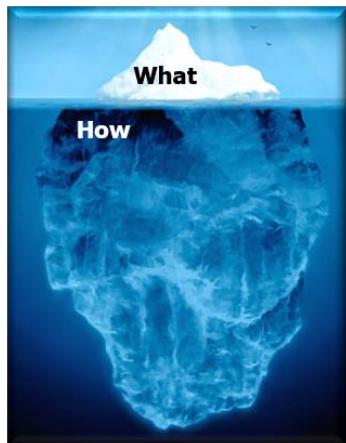
- Java streams provide several key benefits to programs & programmers, e.g.
  - Concise & readable**
    - Declarative paradigm focuses on *what* functions to perform, not *how* to perform them



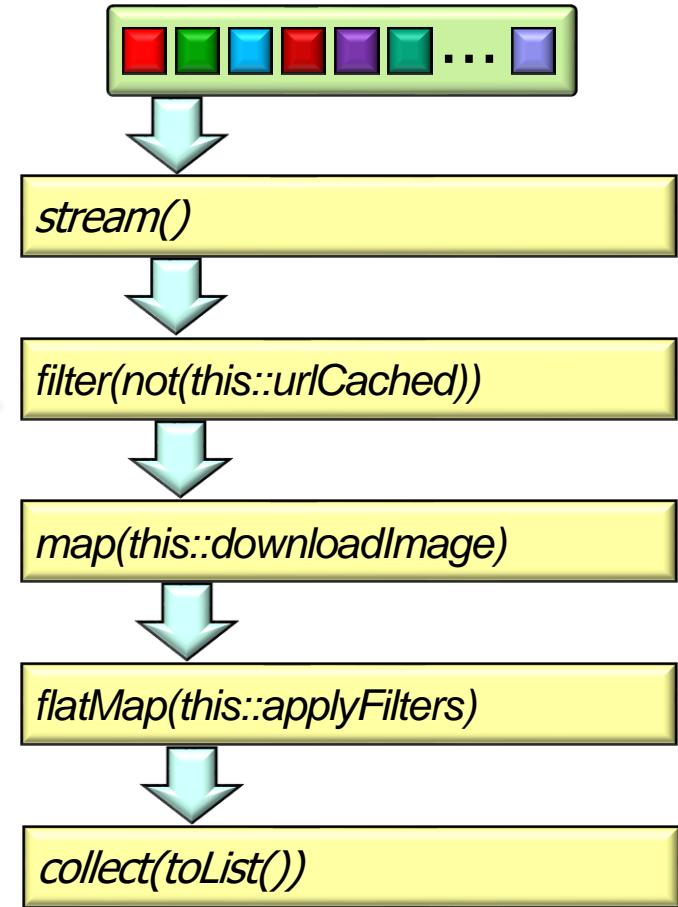
See [en.wikipedia.org/wiki/Declarative\\_programming](https://en.wikipedia.org/wiki/Declarative_programming)

# Benefits of Java Streams

- Java streams provide several key benefits to programs & programmers, e.g.
  - Concise & readable**
    - Declarative paradigm focuses on *what* functions to perform, not *how* to perform them

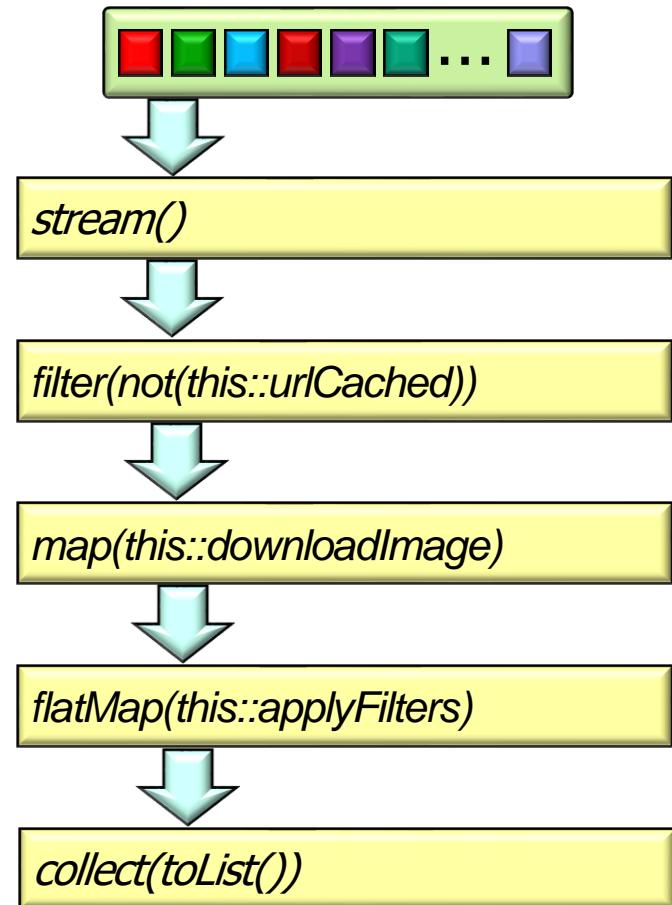
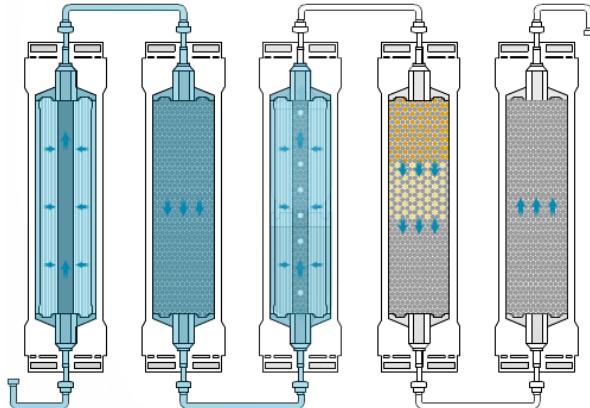


*e.g., no Java control-flow operations are applied in this stream*



# Benefits of Java Streams

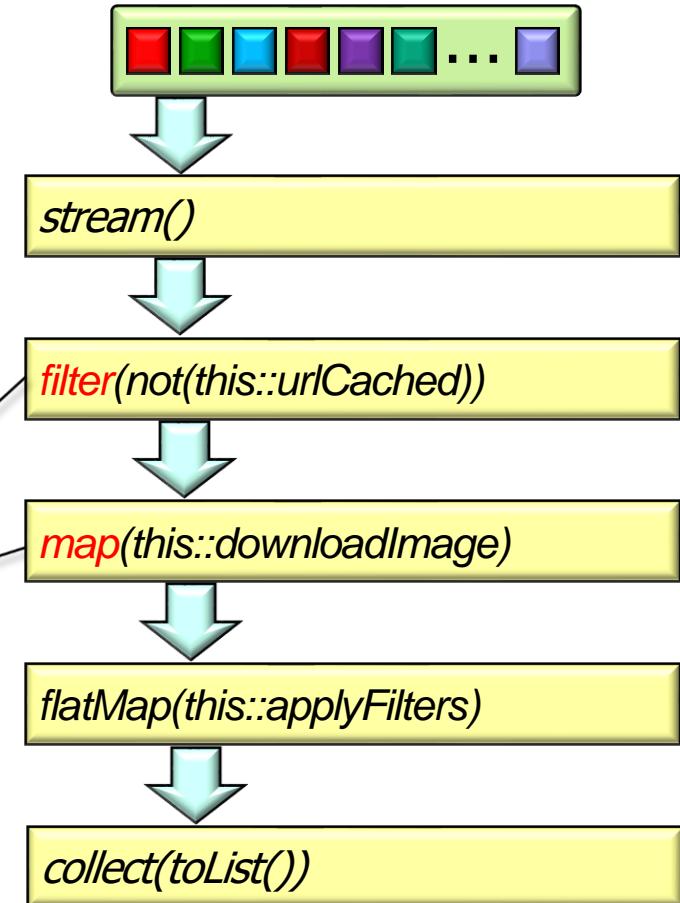
- Java streams provide several key benefits to programs & programmers, e.g.
  - Concise & readable**
  - Flexible & composable**
    - Functions are automatically connected together



# Benefits of Java Streams

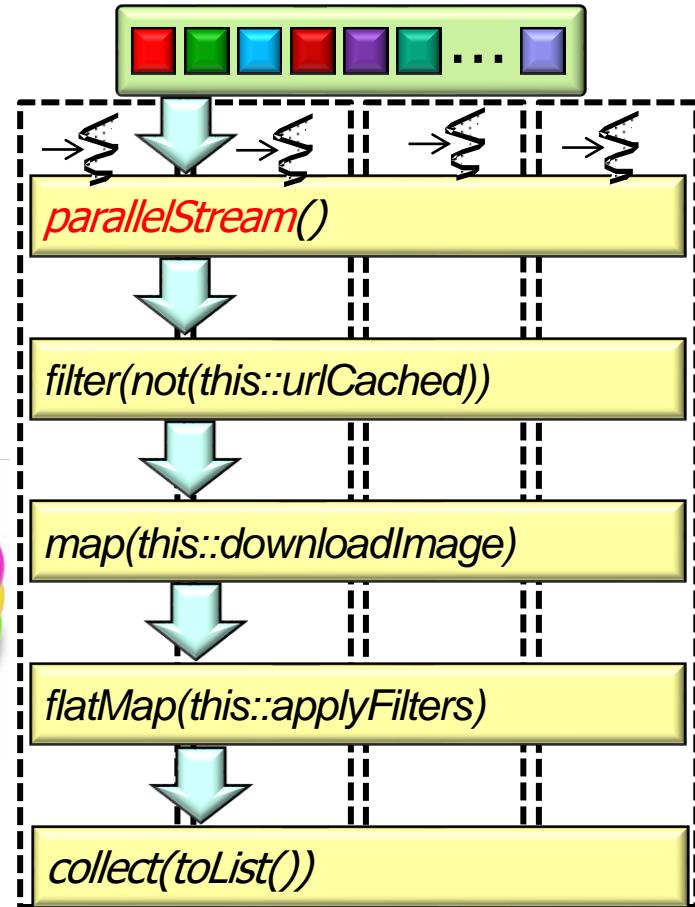
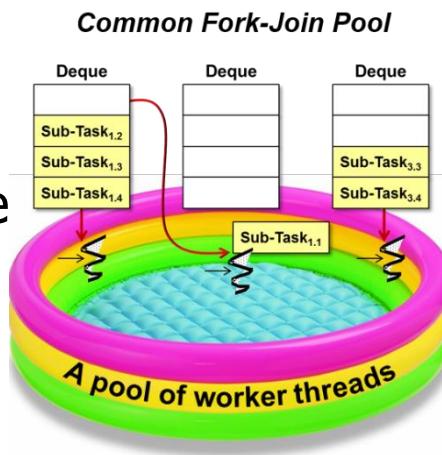
- Java streams provide several key benefits to programs & programmers, e.g.
  - Concise & readable**
  - Flexible & composable**
    - Functions are automatically connected together

*e.g., the output from filter() is passed as the input to map() etc.*



# Benefits of Java Streams

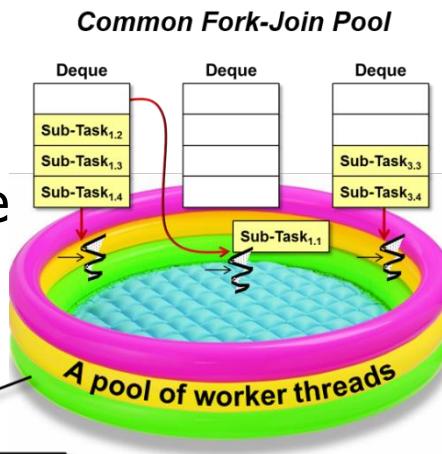
- Java streams provide several key benefits to programs & programmers, e.g.
  - Concise & readable
  - Flexible & composable
  - Simplified scalability**
    - Parallelize performance without the need to write any multi-threaded code



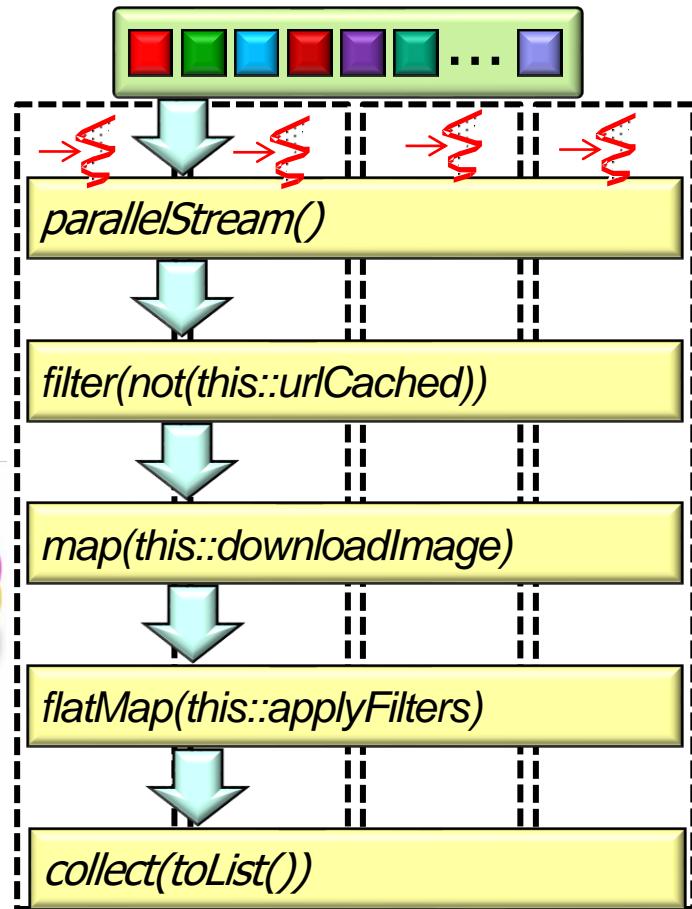
See [docs.oracle.com/javase/tutorial/collections/stream/parallelism.html](https://docs.oracle.com/javase/tutorial/collections/stream/parallelism.html)

# Benefits of Java Streams

- Java streams provide several key benefits to programs & programmers, e.g.
  - Concise & readable
  - Flexible & composable
  - Simplified scalability**
    - Parallelize performance without the need to write any multi-threaded code



*A pool of worker threads is used to process behaviors in parallel*

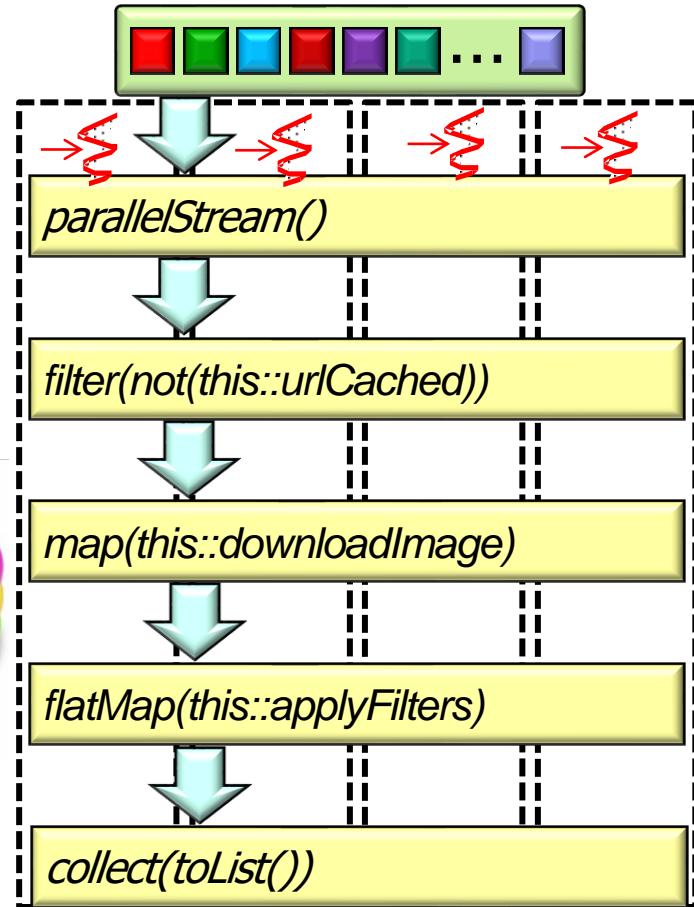
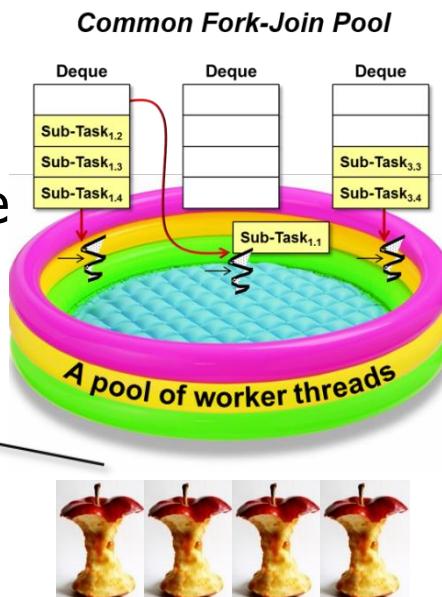


See [dzone.com/articles/common-fork-join-pool-and-streams](https://dzone.com/articles/common-fork-join-pool-and-streams)

# Benefits of Java Streams

- Java streams provide several key benefits to programs & programmers, e.g.
  - Concise & readable
  - Flexible & composable
  - Simplified scalability**
    - Parallelize performance without the need to write any multi-threaded code

*Data mapped automatically to underlying processor cores*



---

# End of Recognize Java Streams Benefits

# **Overview of the Simple SearchStream Program**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

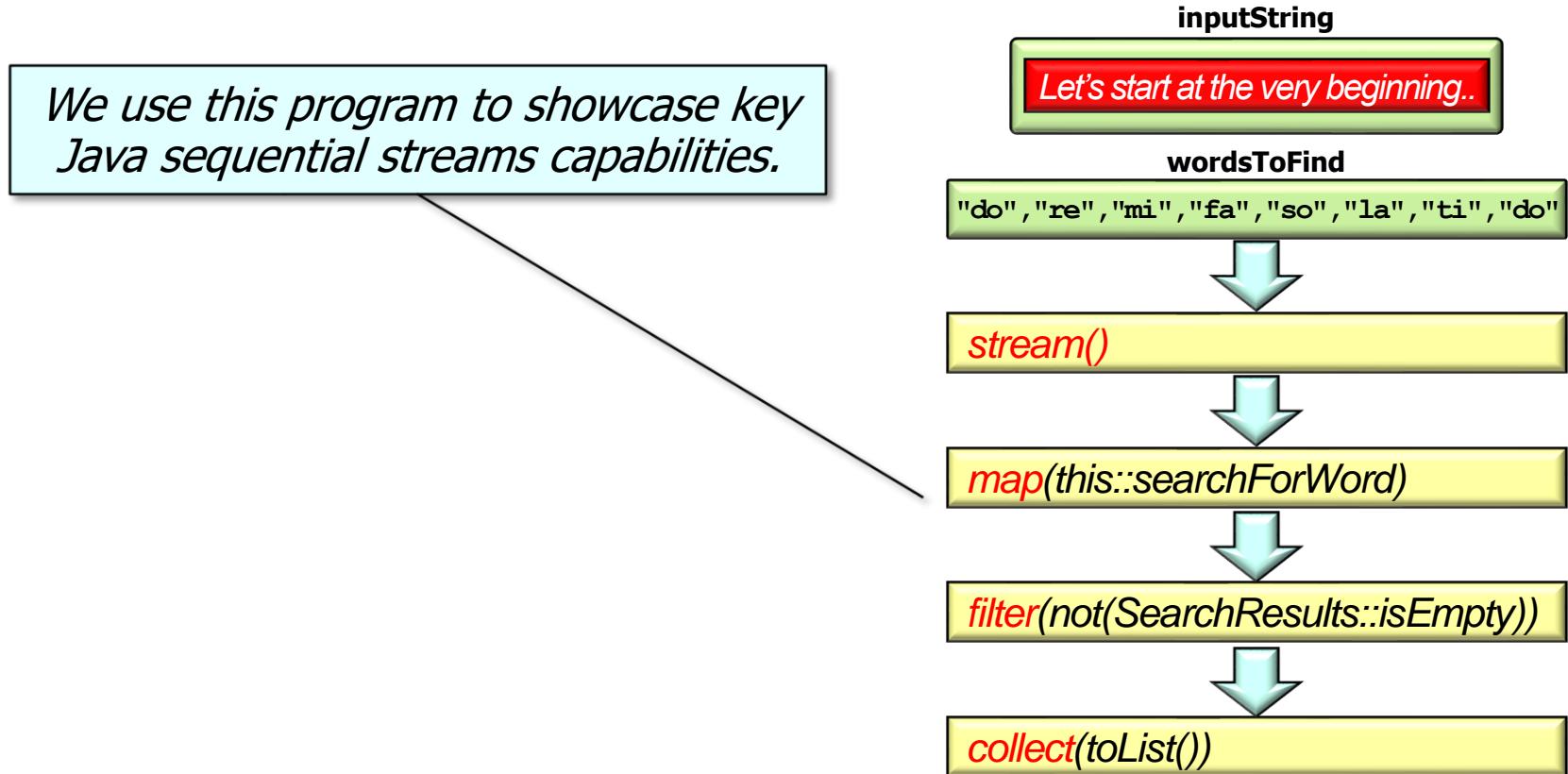
**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of the SimpleSearchStream program

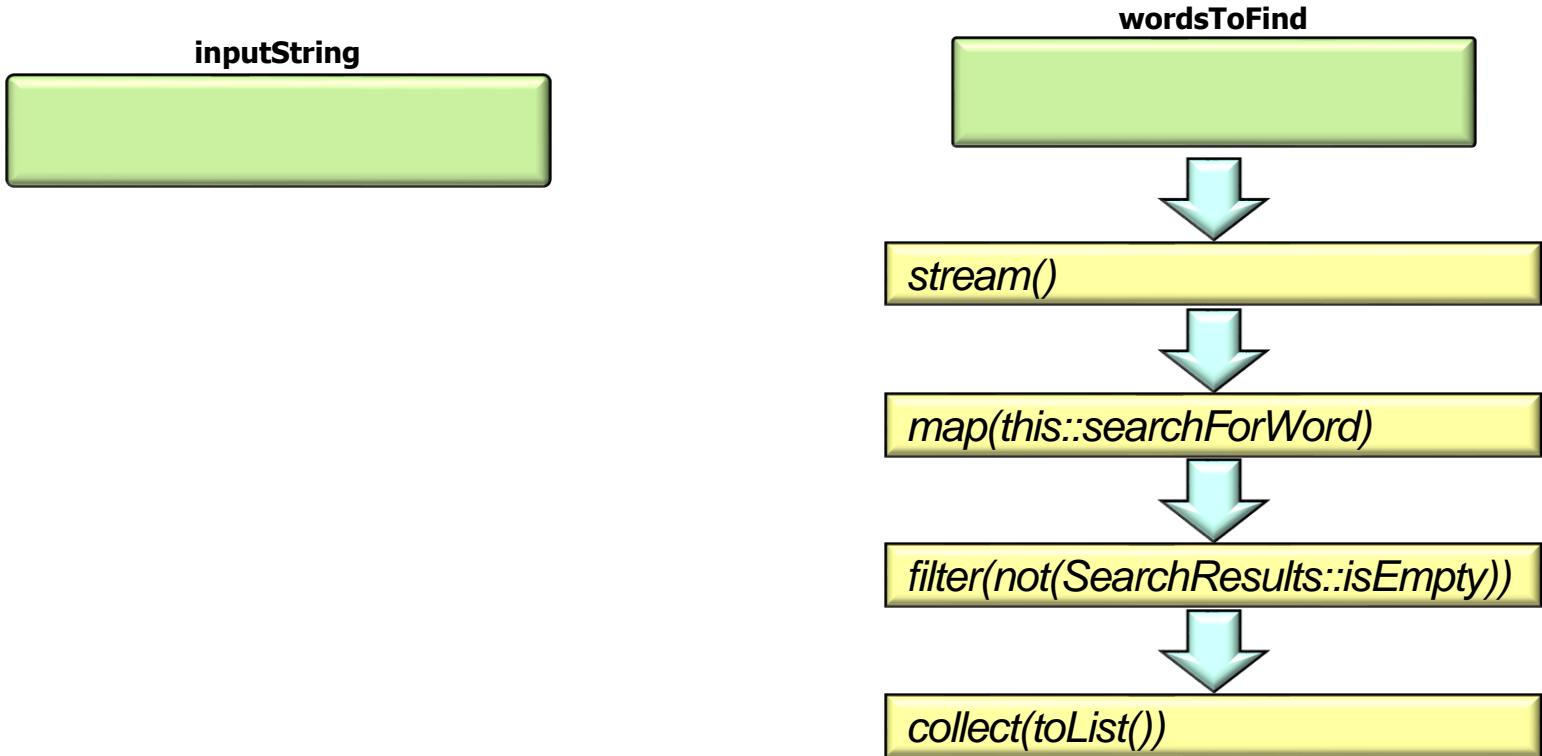


---

# Visualizing the Simple SearchStream program

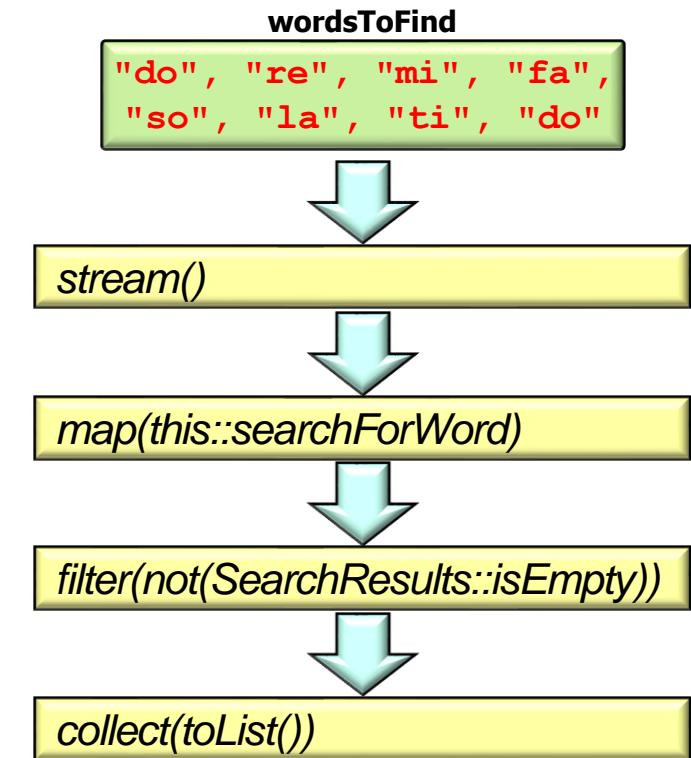
# Visualizing the SimpleSearchStream Program

- This program finds words in an input string



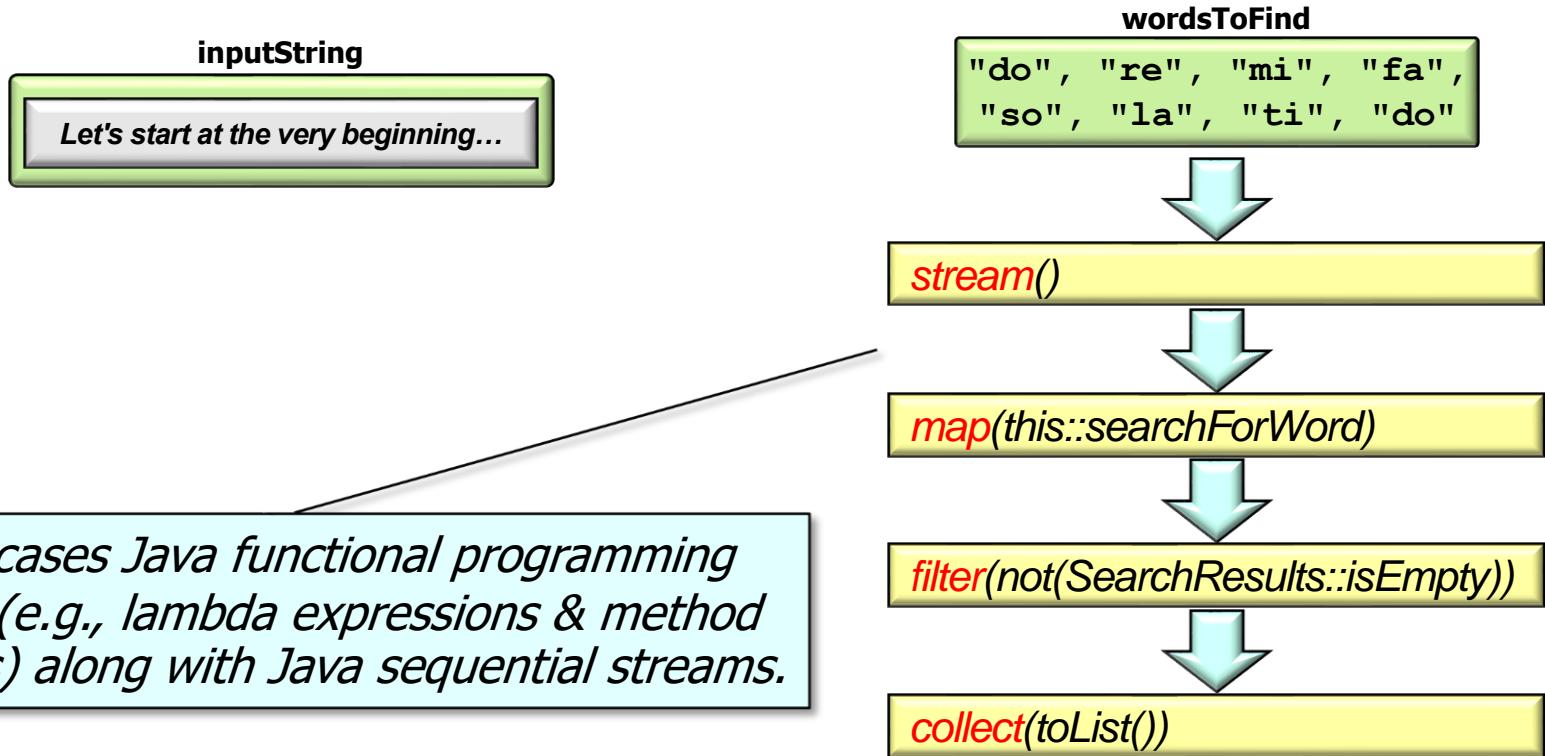
# Visualizing the SimpleSearchStream Program

- This program finds words in an input string



# Visualizing the SimpleSearchStream Program

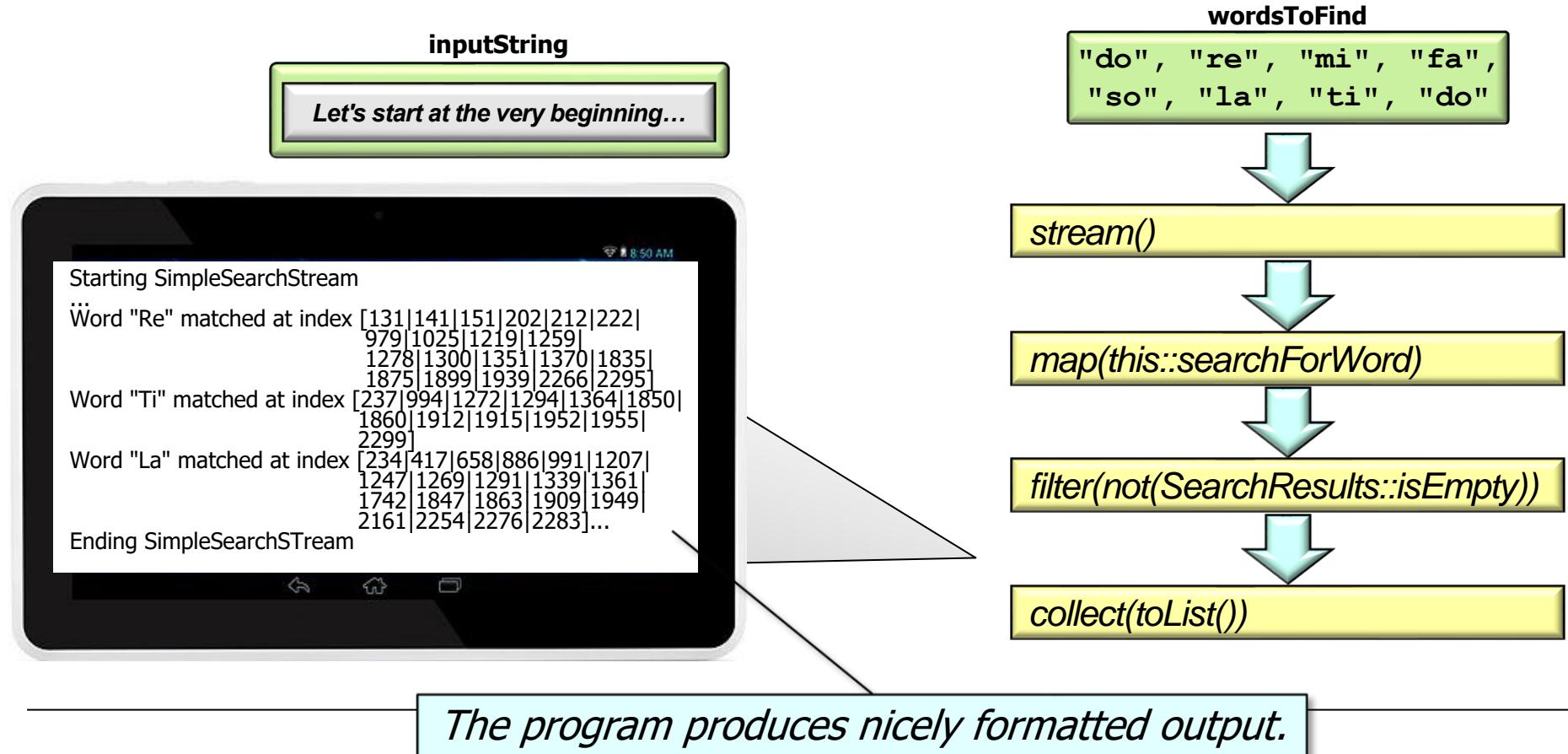
- This program finds words in an input string



See [SimpleSearchStream/src/main/java/search/WordSearcher.java](#)

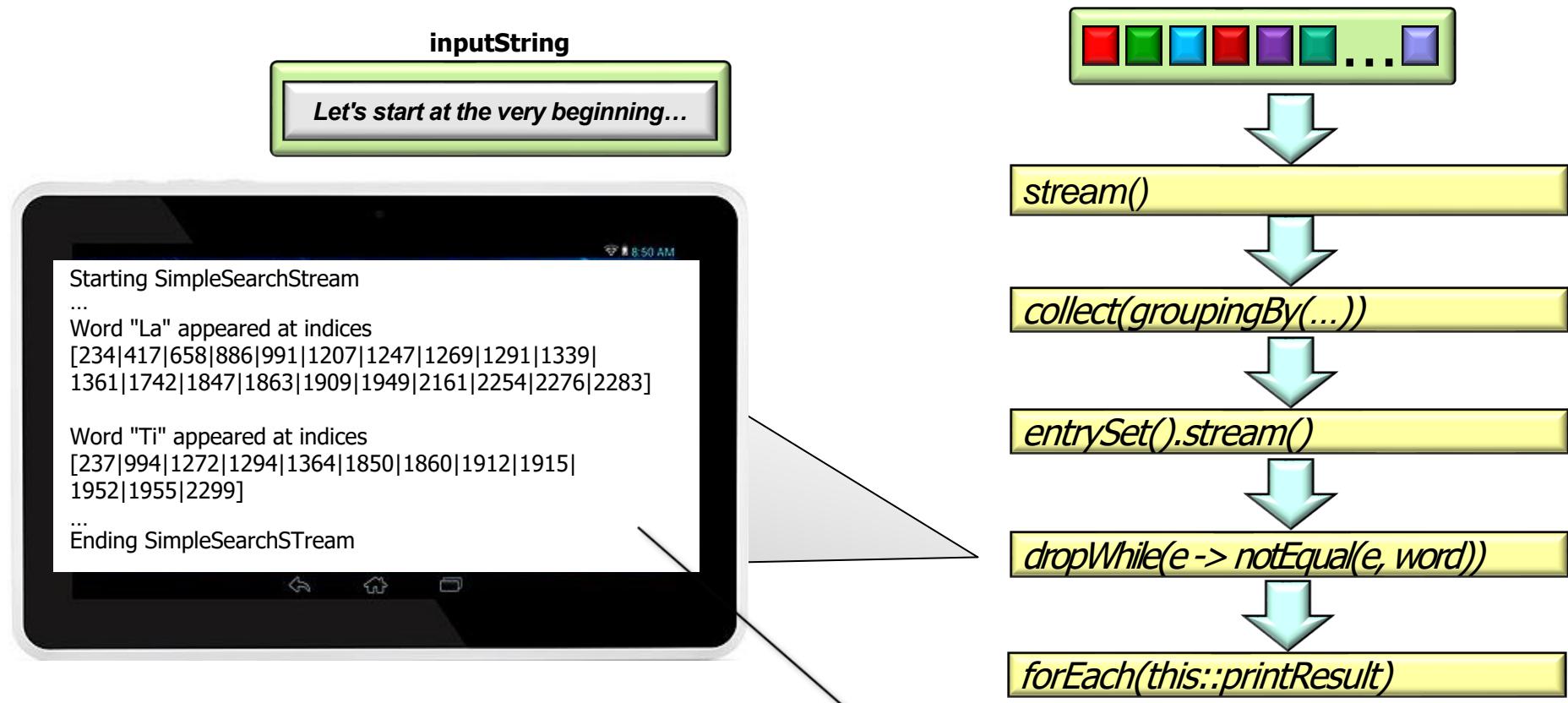
# Visualizing the SimpleSearchStream Program

- This program finds words in an input string



# Visualizing the SimpleSearchStream Program

- It also prints a slice of search results starting at a particular word, e.g., "La"



*Print out results of each map entry (key = word & value = list of search results).*

---

# Entry Point Into the Simple SearchStream Program

# Entry Point Into the SimpleSearchStream Program

---

- It searches sequentially for words in a string containing the contents of a file

```
static public void main(String[] args) { ...
```

```
 String input = TestDataFactory
```

```
 .getInput(sINPUT_FILE, "@") .get(0);
```

```
List<String> wordsToFind = TestDataFactory
```

```
 .getWordList(sWORD_LIST_FILE);
```

```
WordSearcher searcher =
```

```
 new WordSearcher(input);
```

```
List<SearchResults> results =
```

```
 searcher.findWords(wordsToFind);
```

```
searcher.printResults(results); ...
```

---

See [SimpleSearchStream/src/main/java/Main.java](#)

# Entry Point Into the SimpleSearchStream Program

- It searches sequentially for words in a string containing the contents of a file

```
static public void main(String[] args) { ...
```

```
 String input = TestDataFactory
```

```
 .getInput(sINPUT_FILE, "@").get(0);
```

```
List<String> wordsToFind = TestDataFactory
```

```
 .getWordList(sWORD_LIST_FILE);
```

```
WordSearcher searcher =
```

```
 new WordSearcher(input);
```

```
List<SearchResults> results =
```

```
 searcher.findWords(wordsToFind);
```

```
searcher.printResults(results); ...
```

*Create an input string containing  
the lyrics to the do-re-mi song.*

See [SimpleSearchStream/src/main/java/utils/TestDataFactory.java](#)

# Entry Point Into the SimpleSearchStream Program

- It searches sequentially for words in a string containing the contents of a file

```
static public void main(String[] args) { ...
```

```
 String input = TestDataFactory
```

```
 .getInput(sINPUT_FILE, "@") .get(0);
```

```
List<String> wordsToFind = TestDataFactory
 .getWordList(sWORD_LIST_FILE);
```

```
WordSearcher searcher =
 new WordSearcher(input);
```

*Get the list of words to find.*

```
List<SearchResults> results =
 searcher.findWords(wordsToFind);
```

```
searcher.printResults(results); ...
```

See [SimpleSearchStream/src/main/java/utils/TestDataFactory.java](#)

# Entry Point Into the SimpleSearchStream Program

- It searches sequentially for words in a string containing the contents of a file

```
static public void main(String[] args) { ...
```

```
 String input = TestDataFactory
```

```
 .getInput(sINPUT_FILE, "@") .get(0);
```

```
List<String> wordsToFind = TestDataFactory
```

```
 .getWordList(sWORD_LIST_FILE);
```

```
WordSearcher searcher =
```

```
 new WordSearcher(input);
```

*Create an object used to search  
for words in the input string.*

```
List<SearchResults> results =
```

```
 searcher.findWords(wordsToFind);
```

```
searcher.printResults(results); ...
```

See [SimpleSearchStream/src/main/java/search/WordSearcher.java](#)

# Entry Point Into the SimpleSearchStream Program

- It searches sequentially for words in a string containing the contents of a file

```
static public void main(String[] args) { ...
```

```
 String input = TestDataFactory
```

```
 .getInput(sINPUT_FILE, "@") .get(0);
```

```
List<String> wordsToFind = TestDataFactory
```

```
 .getWordList(sWORD_LIST_FILE);
```

```
WordSearcher searcher =
 new WordSearcher(input);
```

*Find all matching words.*

```
List<SearchResults> results =
 searcher.findWords(wordsToFind);
```

```
searcher.printResults(results); ...
```

# Entry Point Into the SimpleSearchStream Program

- It searches sequentially for words in a string containing the contents of a file

```
static public void main(String[] args) { ...
```

```
 String input = TestDataFactory
```

```
 .getInput(sINPUT_FILE, "@") .get(0);
```

```
List<String> wordsToFind = TestDataFactory
```

```
 .getWordList(sWORD_LIST_FILE);
```

```
WordSearcher searcher =
```

```
 new WordSearcher(input);
```

```
List<SearchResults> results =
```

```
 searcher.findWords(wordsToFind);
```

*Print all matching words.*

```
searcher.printResults(results); ...
```

---

# End of Overview of the SimpleSearchStream Program

# Visualize the WordSearcher

## .findWords() Method

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

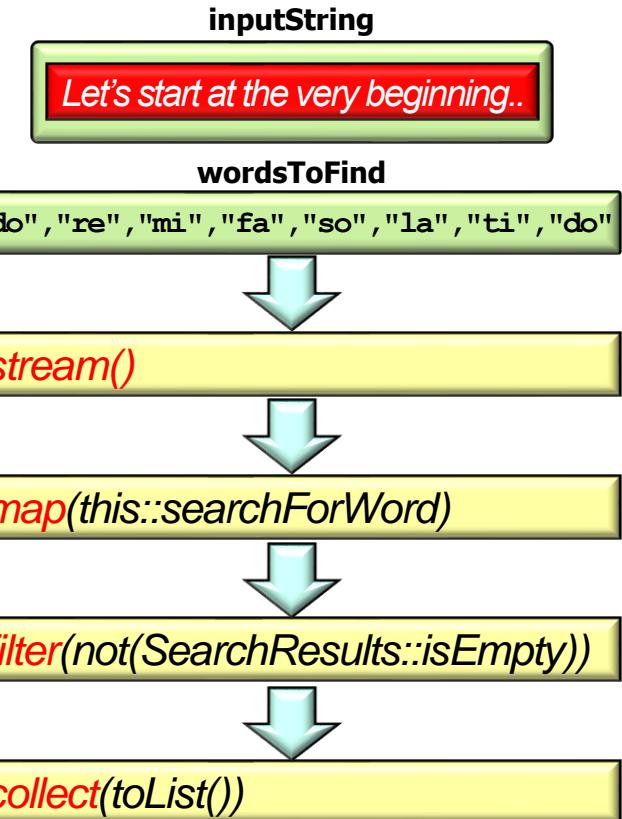
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

- Recognize the structure & functionality of the SimpleSearchStream example
- Visualize aggregate operations in SimpleSearch Stream's WordSearcher.findWords() method



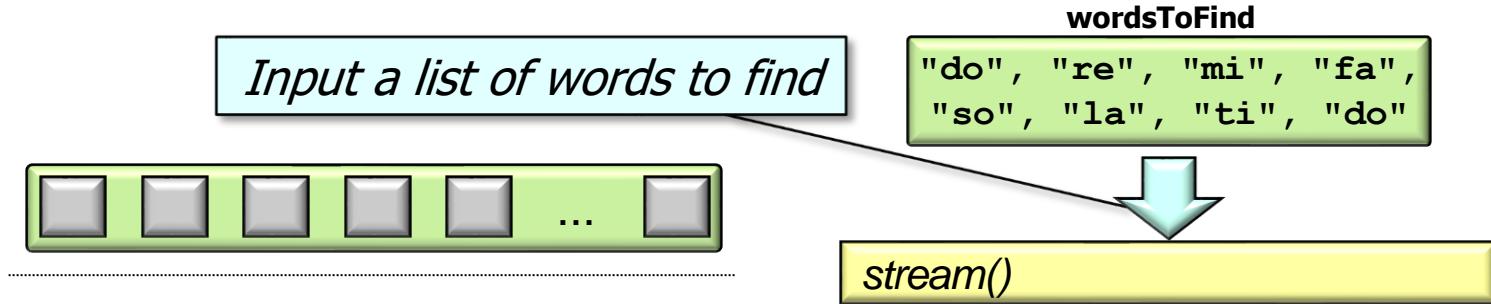
---

# Visualizing the Word Searcher.findWords() Method

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>



See [SimpleSearchStream/src/main/java/search/WordSearcher.java](#)

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>



**wordsToFind**  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

*stream()*



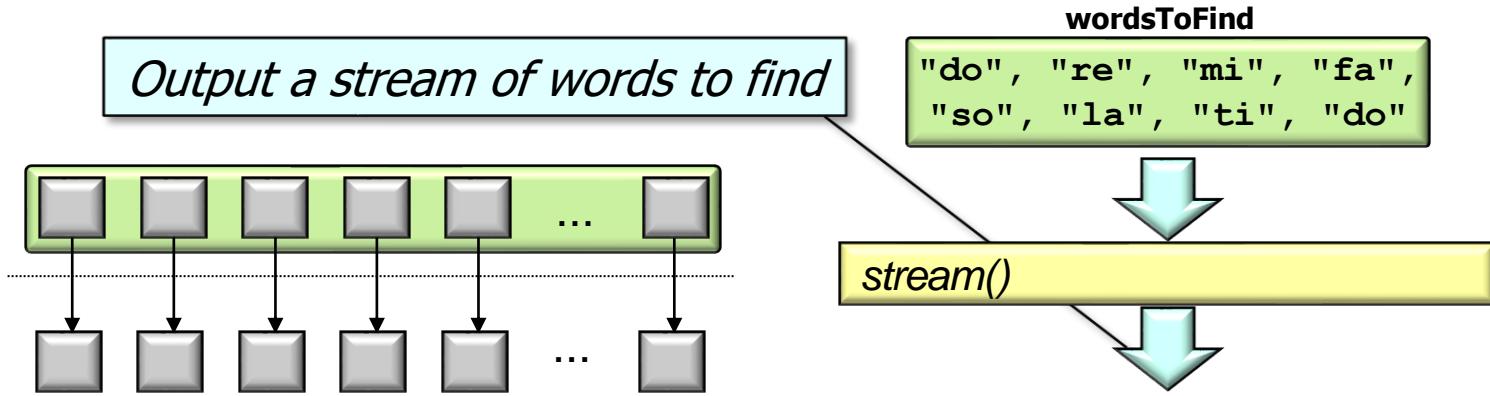
Convert collection to a (sequential) stream

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

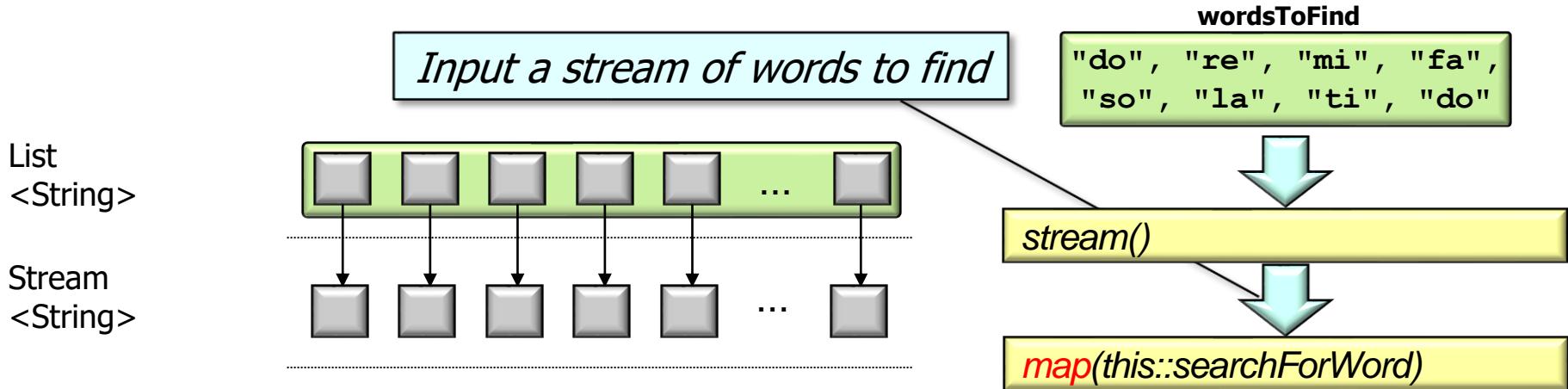
List  
<String>

Stream  
<String>



# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

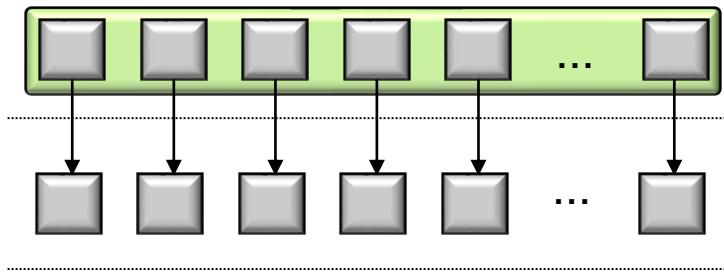


# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>



**wordsToFind**  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

*stream()*

*map(this::searchForWord)*

Search for the word in the input string

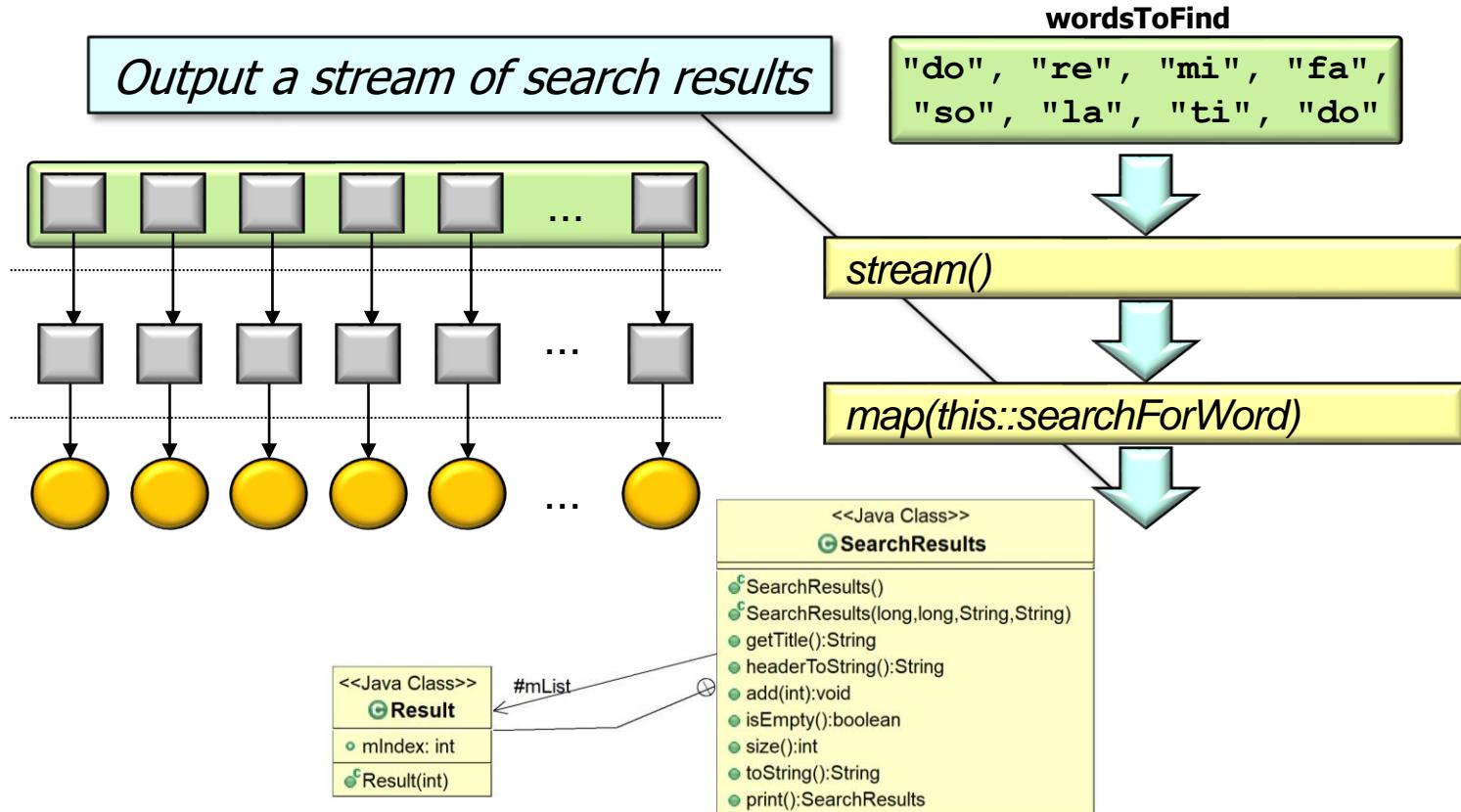
# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>

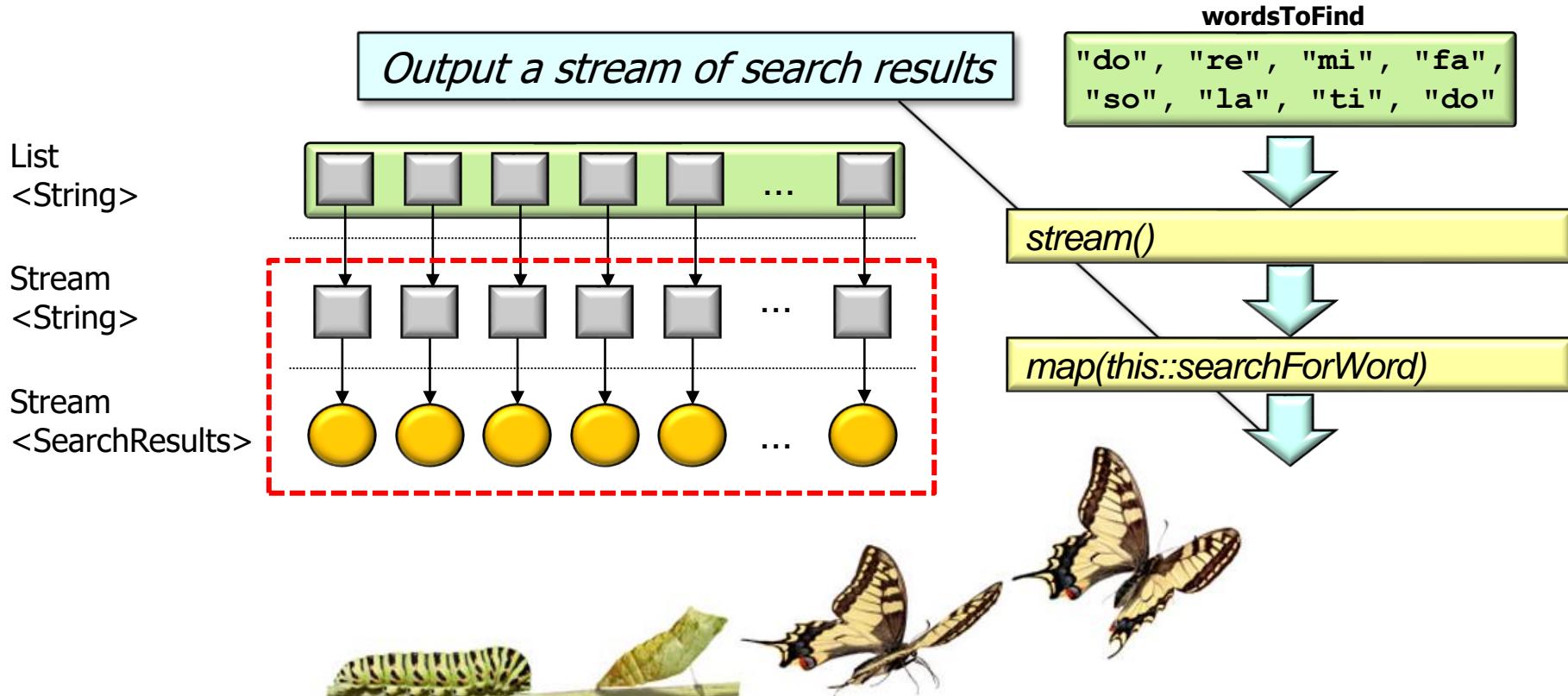
Stream  
<SearchResults>



SearchResults stores # of times a word appeared in the input string

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string



Note the transformation of types at this point in the stream!

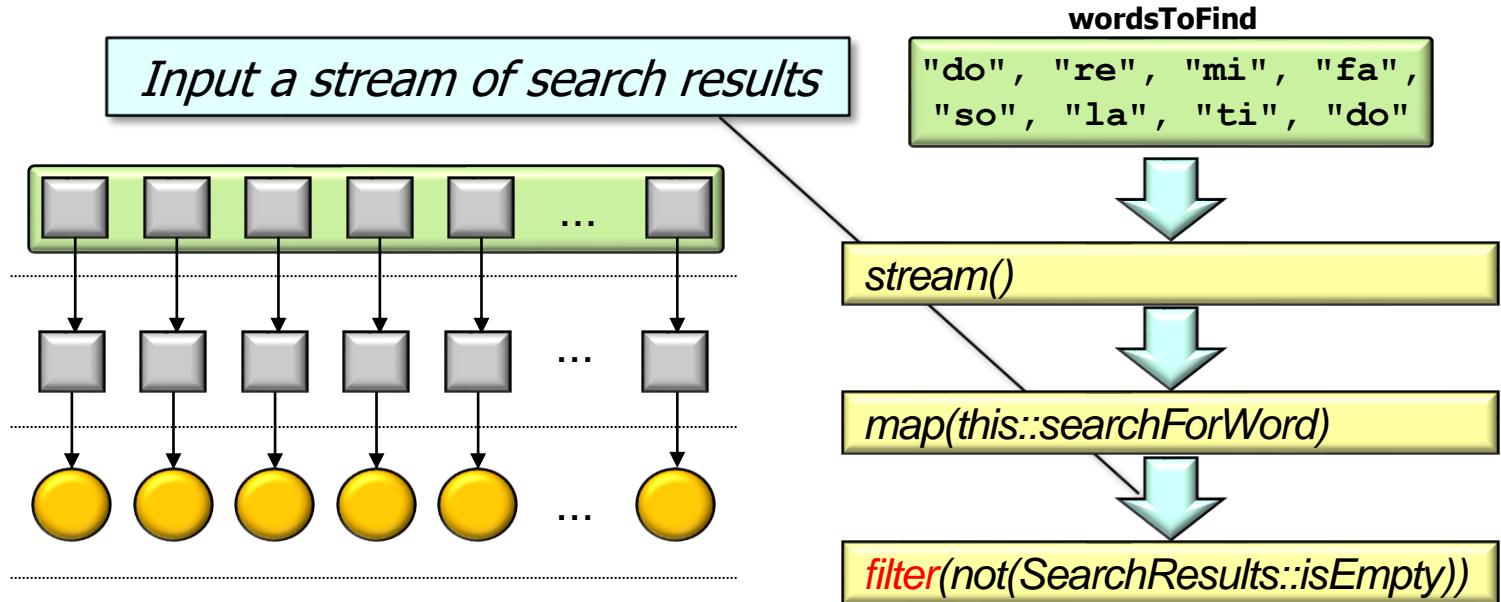
# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>

Stream  
<SearchResults>



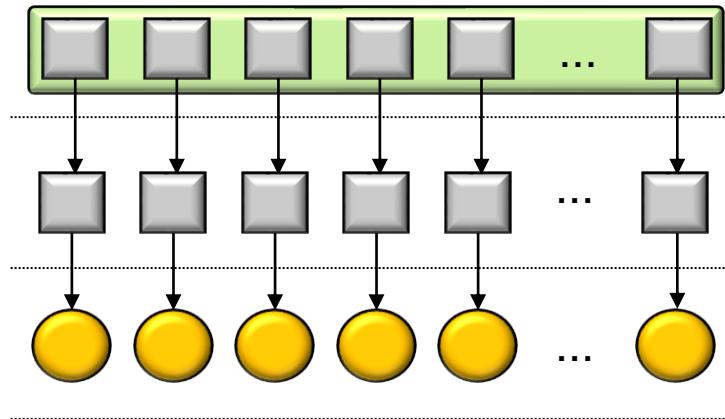
# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>

Stream  
<SearchResults>



wordsToFind  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

Remove empty search results from the stream

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

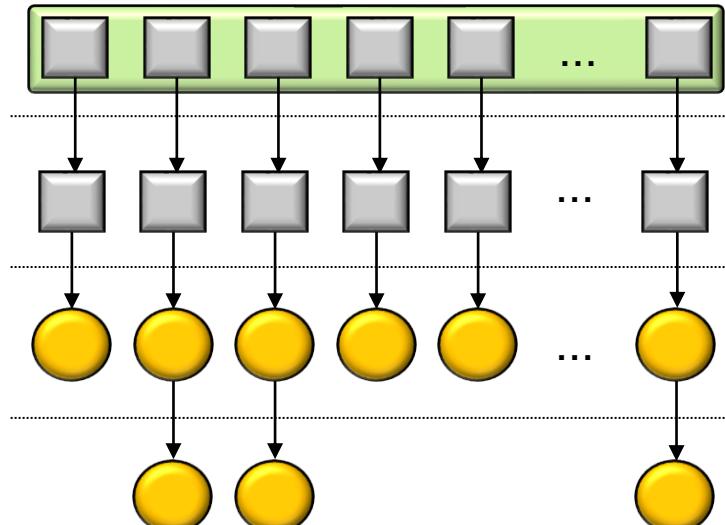
*Output a stream of non-empty search results*

List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>



**wordsToFind**  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

*stream()*

*map(this::searchForWord)*

*filter(not(SearchResults::isEmpty))*

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

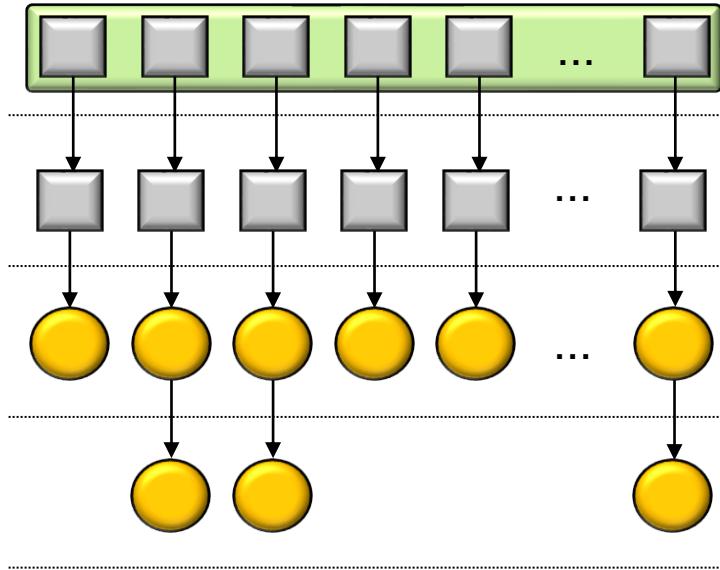
List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>

*Input a stream of non-empty search results*



wordsToFind  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

collect(toList())

# Visualizing the WordSearcher.findWords() Method

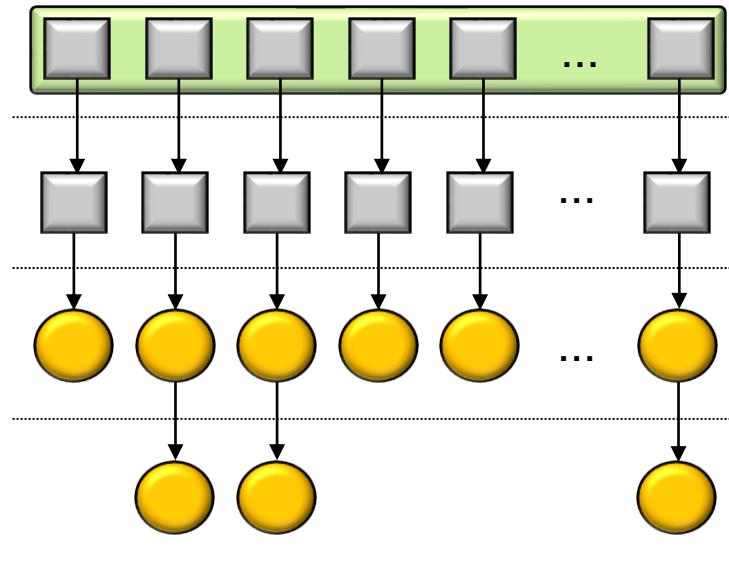
- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>



wordsToFind  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

collect(toList())

Trigger intermediate operation processing

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

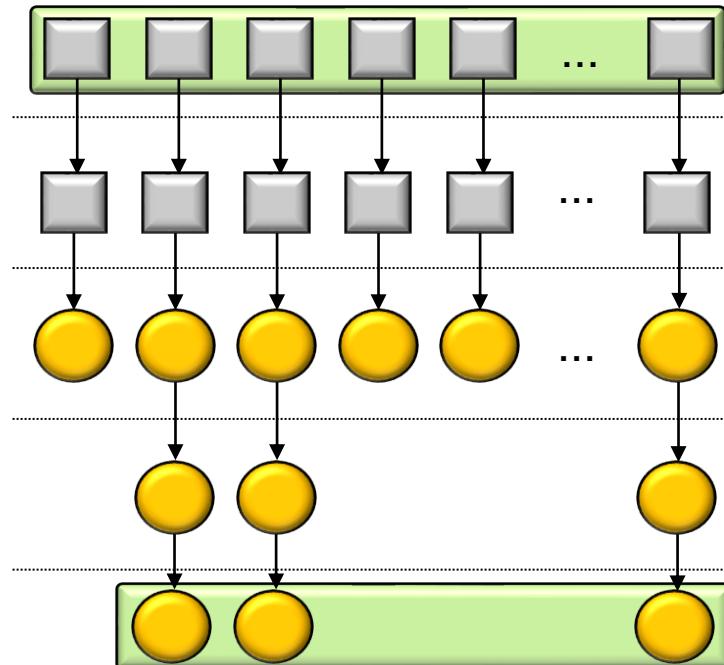
List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>

List  
<SearchResults>



**wordsToFind**  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

collect(toList())

Return a list of non-empty search results

# Visualizing the WordSearcher.findWords() Method

- The “physical” processing of a stream differs from the “logical” model

*It may appear that each “row” of data is processed from “left to right”*

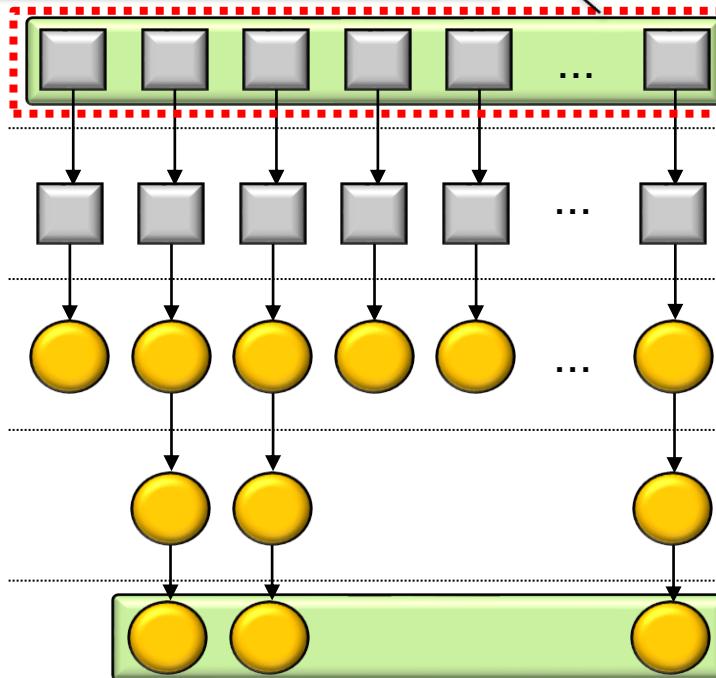
List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>

List  
<SearchResults>



wordsToFind  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

collect(toList())

# Visualizing the WordSearcher.findWords() Method

- The “physical” processing of a stream differs from the “logical” model

*However, each element is actually “pulled” from the source through each aggregate operation*

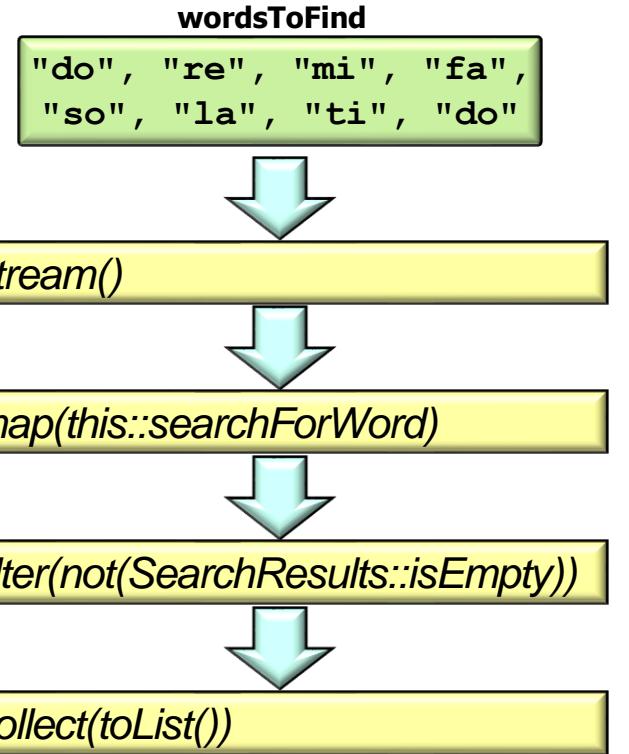
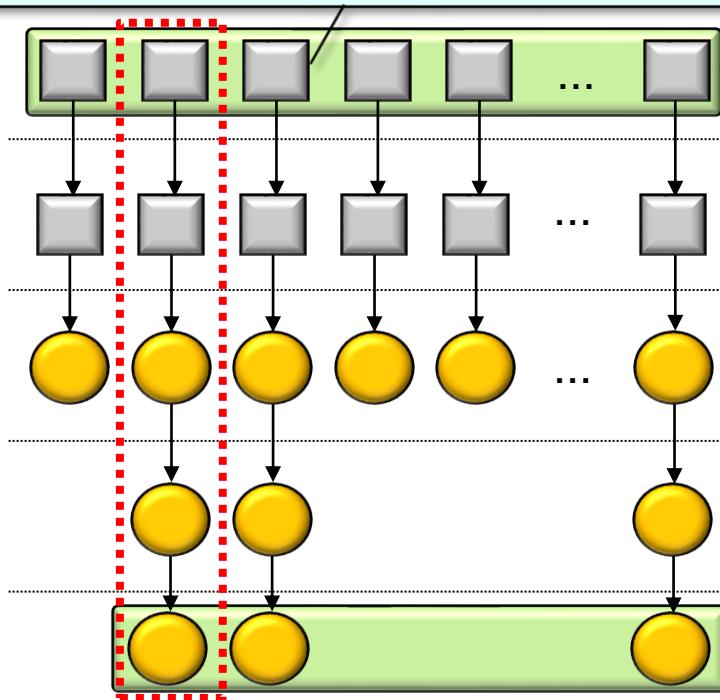
List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>

List  
<SearchResults>



This implementation is much more efficient & supports “short-circuit” operations

---

End of Visualize the Word  
Searcher.findWords() Method

# Visualize the WordSearcher

## .findWords() Method

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

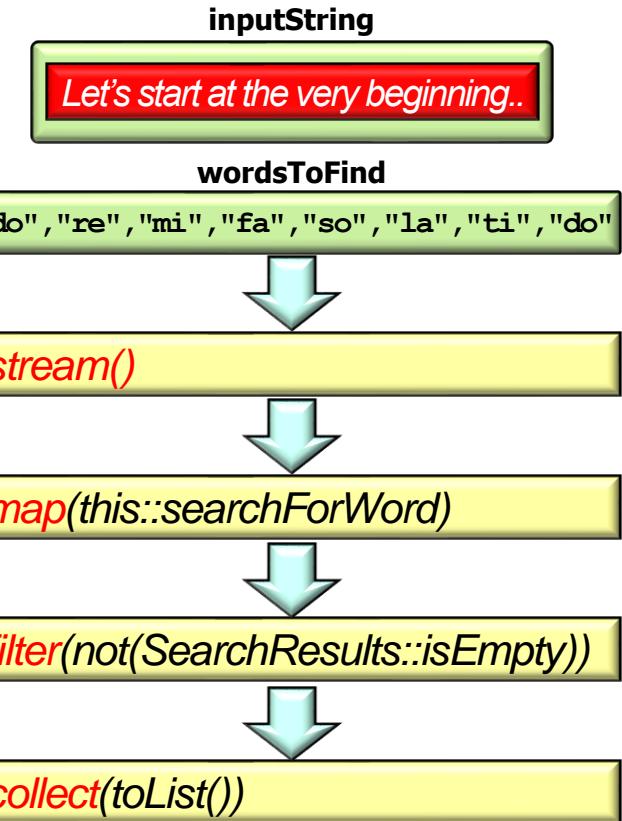
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

- Recognize the structure & functionality of the SimpleSearchStream example
- Visualize aggregate operations in SimpleSearch Stream's WordSearcher.findWords() method



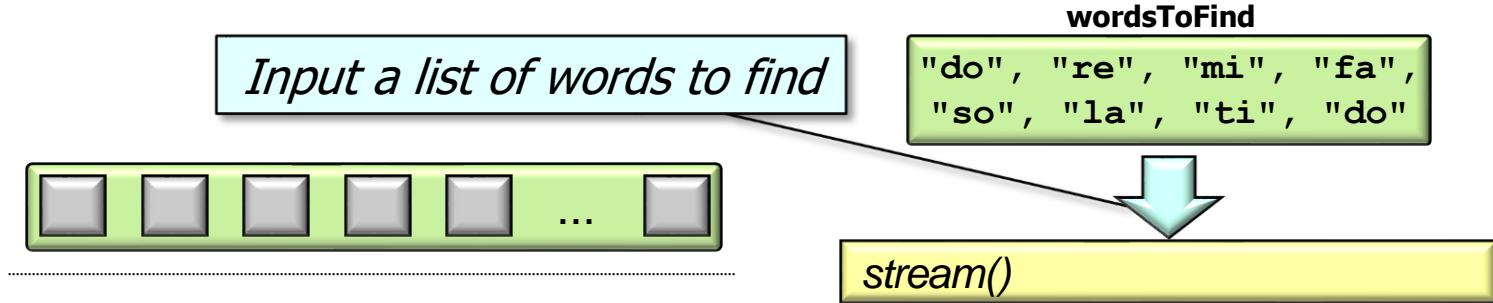
---

# Visualizing the Word Searcher.findWords() Method

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>



See [SimpleSearchStream/src/main/java/search/WordSearcher.java](#)

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>



`wordsToFind`

```
"do", "re", "mi", "fa",
"so", "la", "ti", "do"
```

*stream()*



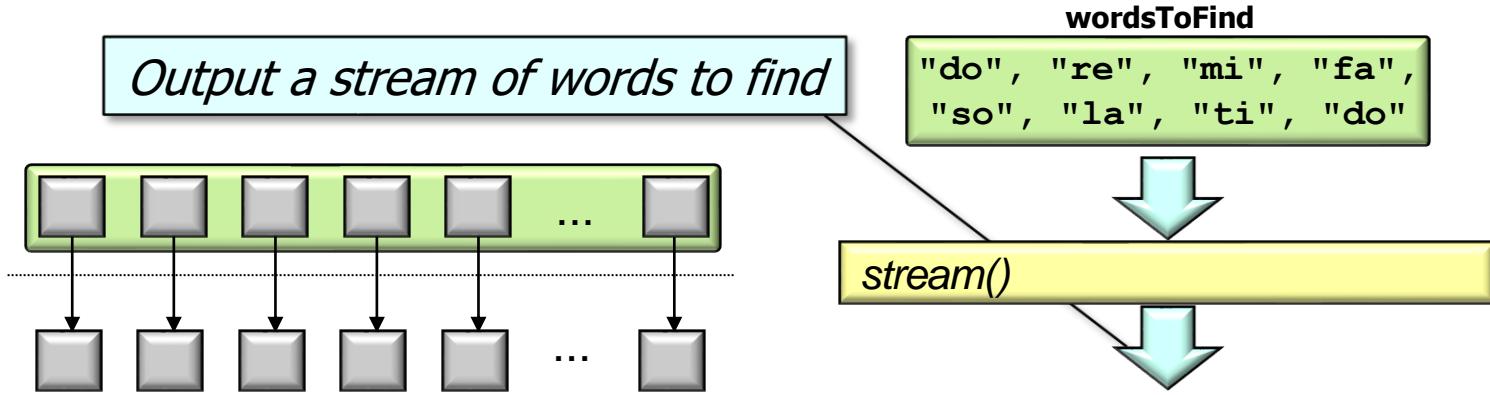
Convert collection to a (sequential) stream

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

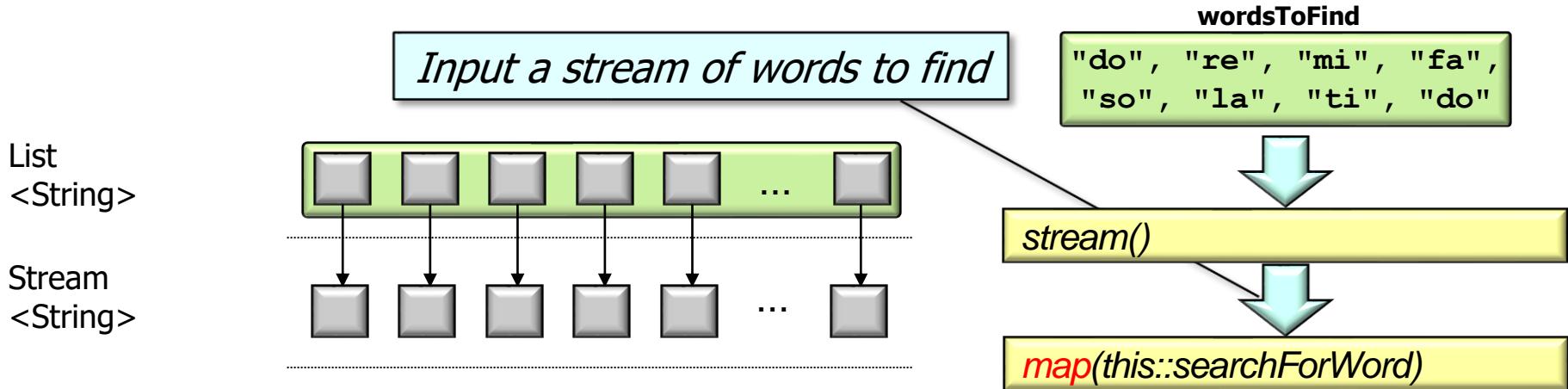
List  
<String>

Stream  
<String>



# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

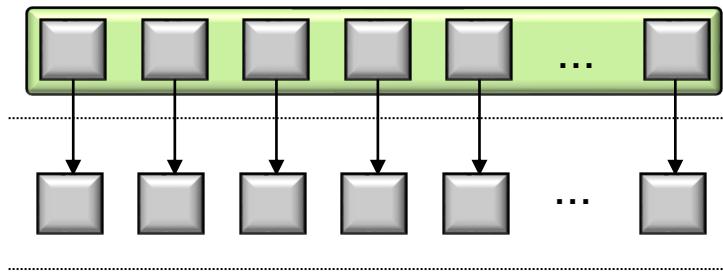


# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>



**wordsToFind**  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

*stream()*

*map(this::searchForWord)*

Search for the word in the input string

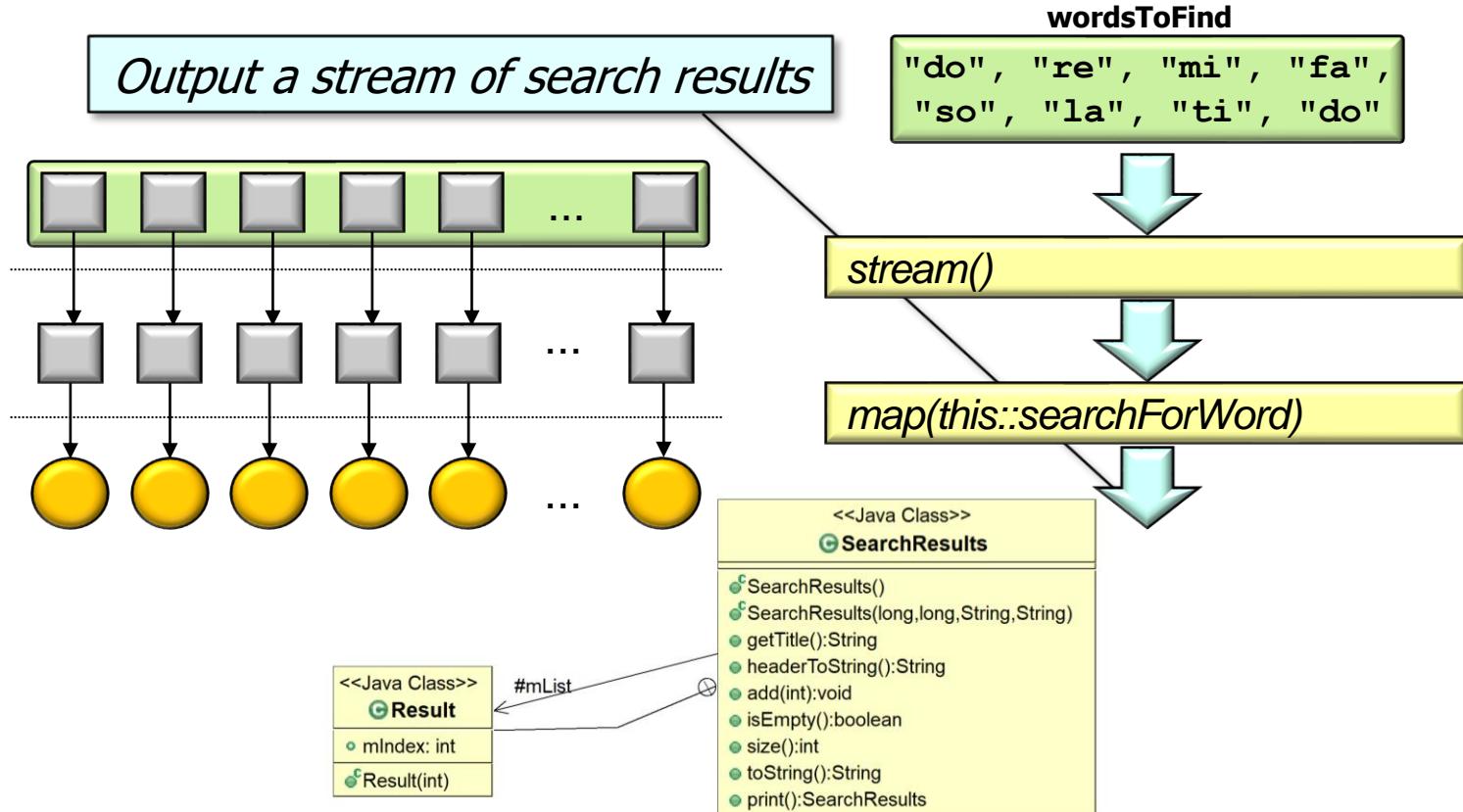
# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>

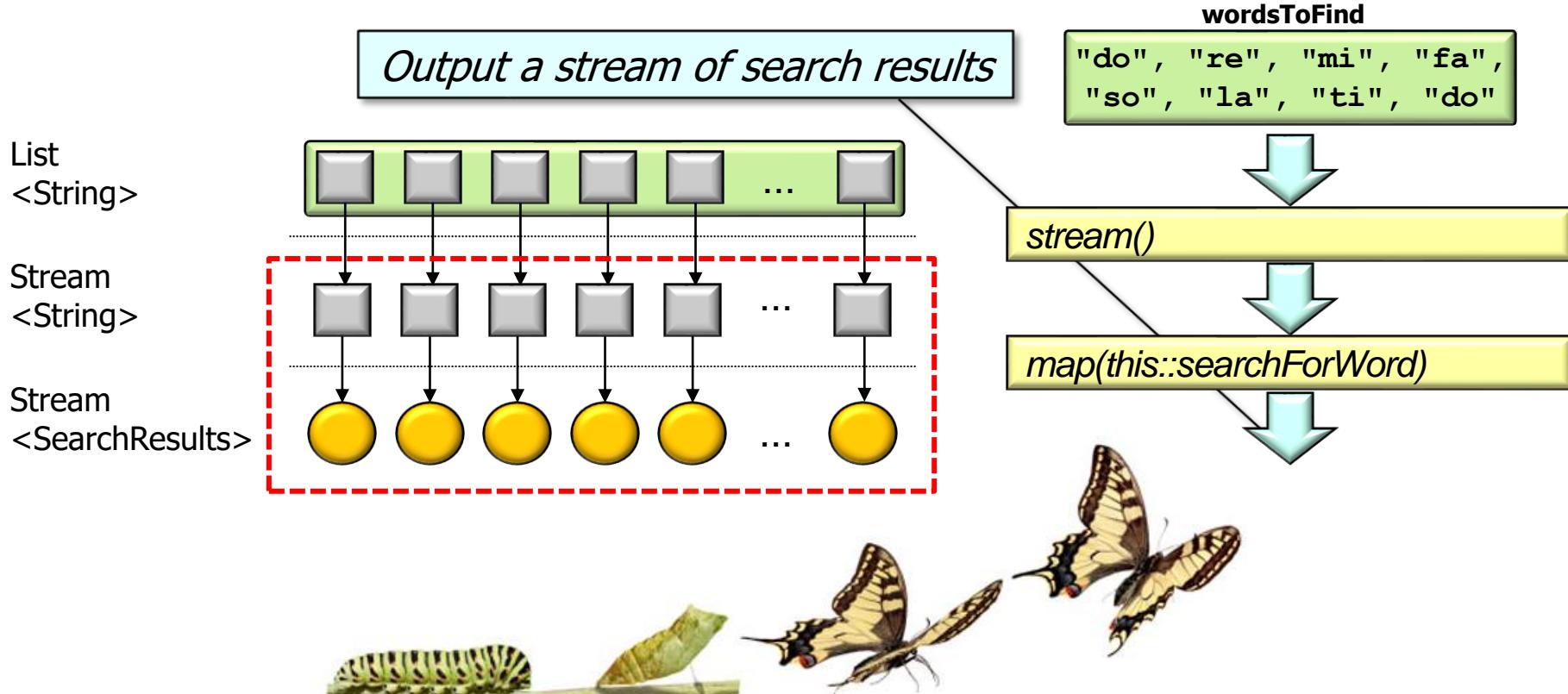
Stream  
<SearchResults>



SearchResults stores # of times a word appeared in the input string

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string



Note the transformation of types at this point in the stream!

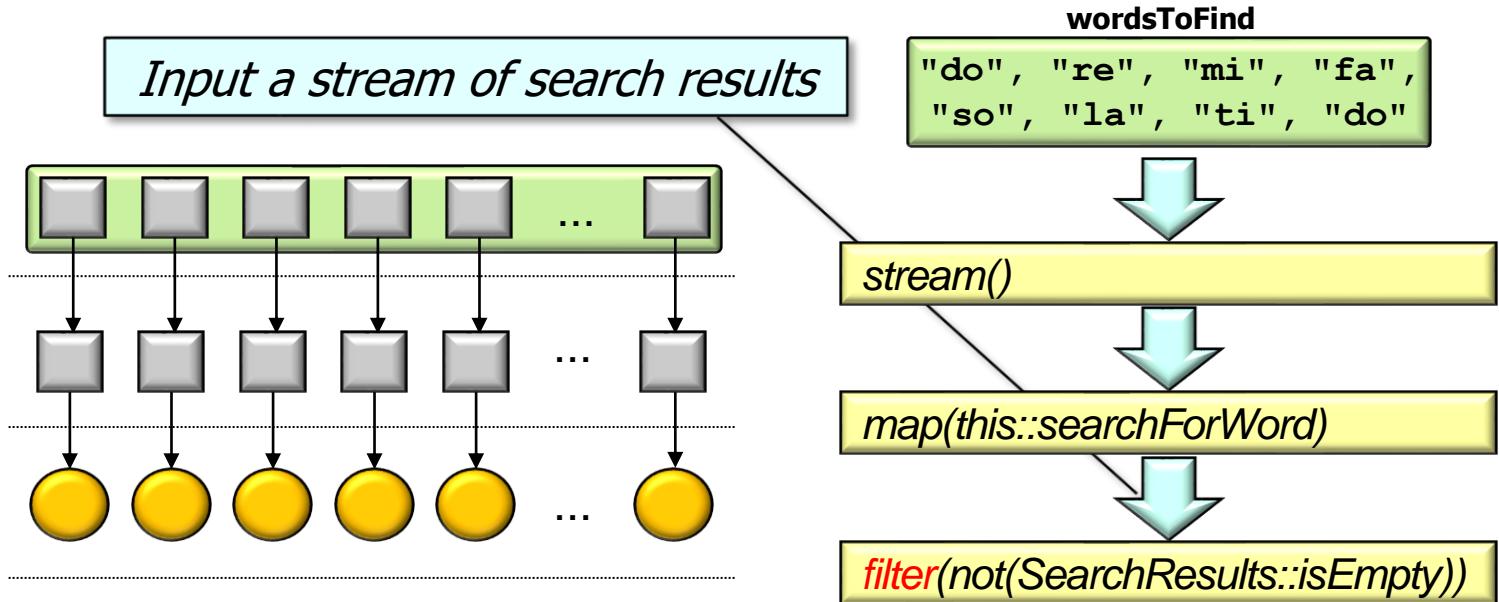
# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>

Stream  
<SearchResults>



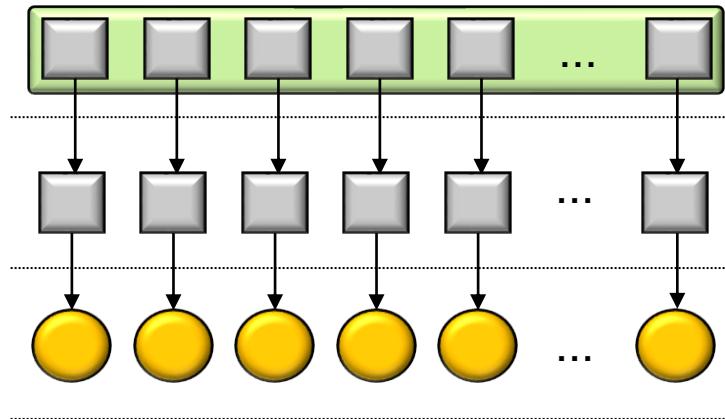
# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>

Stream  
<SearchResults>



wordsToFind  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

Remove empty search results from the stream

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

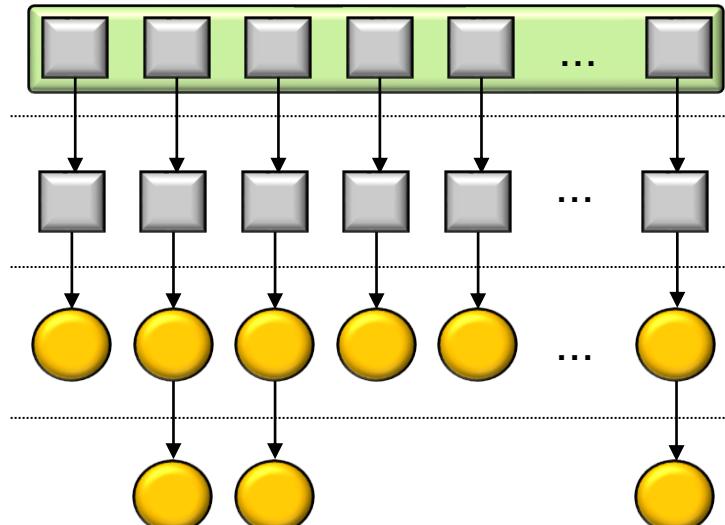
*Output a stream of non-empty search results*

List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>



**wordsToFind**  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

*stream()*

*map(this::searchForWord)*

*filter(not(SearchResults::isEmpty))*

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

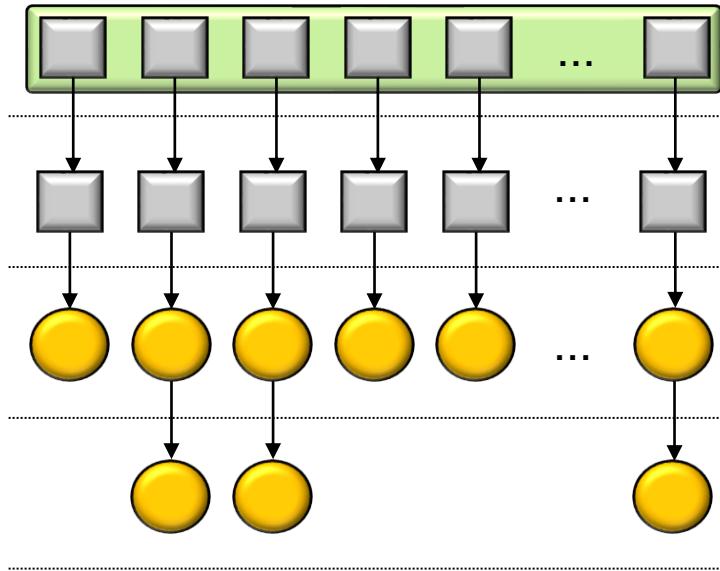
List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>

*Input a stream of non-empty search results*



wordsToFind  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

collect(toList())

# Visualizing the WordSearcher.findWords() Method

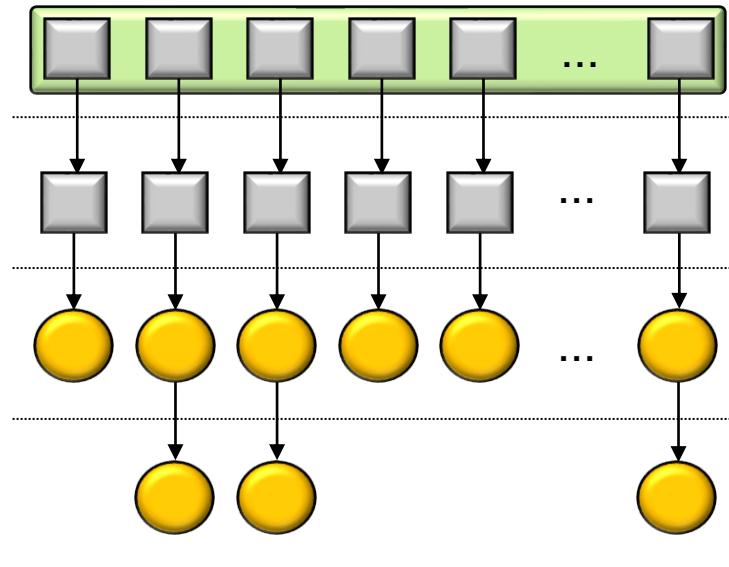
- WordSearcher.findWords() searches for words in an input string

List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>



wordsToFind  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

collect(toList())

Trigger intermediate operation processing

# Visualizing the WordSearcher.findWords() Method

- WordSearcher.findWords() searches for words in an input string

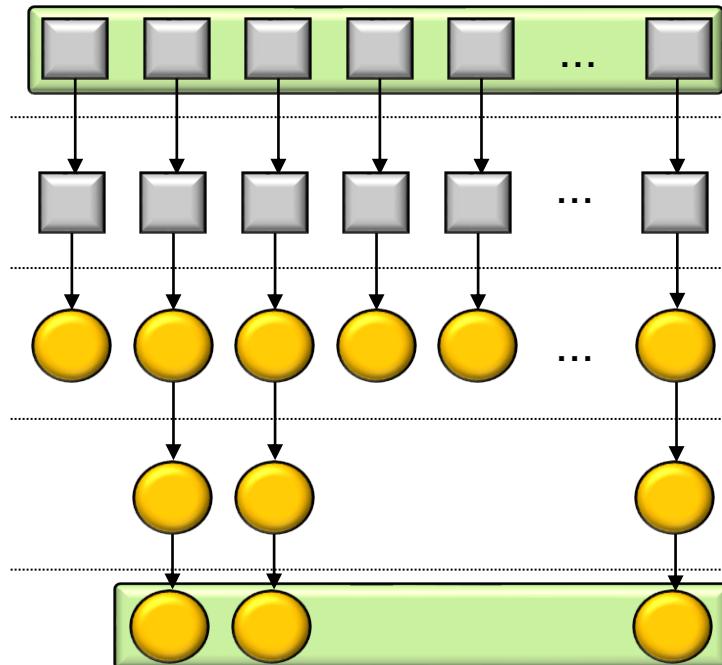
List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>

List  
<SearchResults>



**wordsToFind**  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

collect(toList())

Return a list of non-empty search results

# Visualizing the WordSearcher.findWords() Method

- The “physical” processing of a stream differs from the “logical” model

*It may appear that each “row” of data is processed from “left to right”*

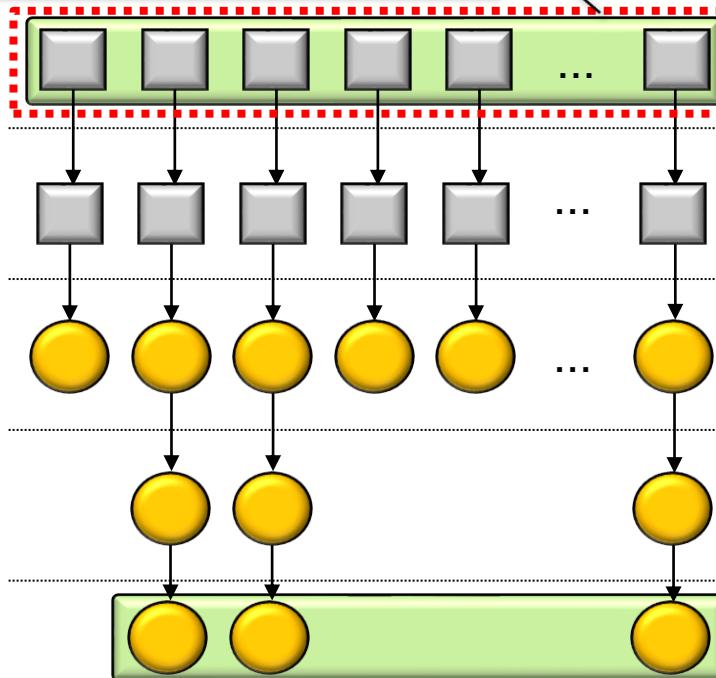
List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>

List  
<SearchResults>



wordsToFind  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

collect(toList())

# Visualizing the WordSearcher.findWords() Method

- The “physical” processing of a stream differs from the “logical” model

*However, each element is actually “pulled” from the source through each aggregate operation*

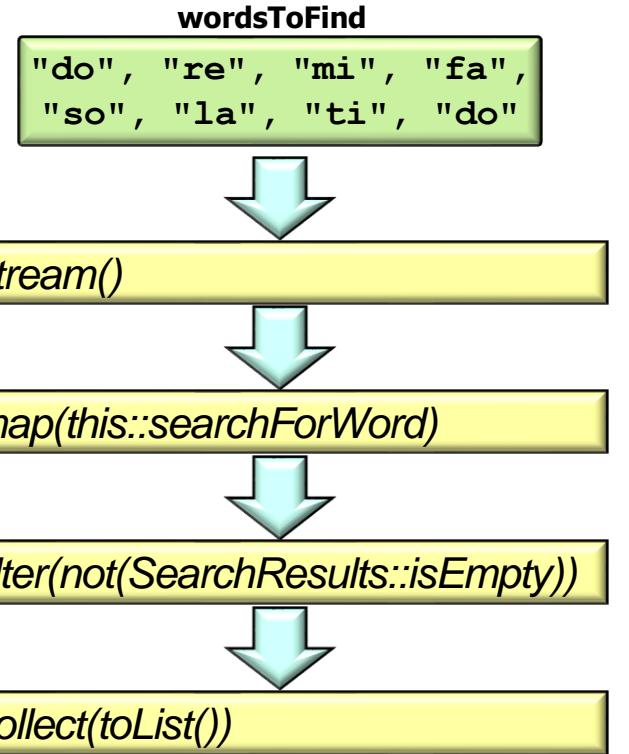
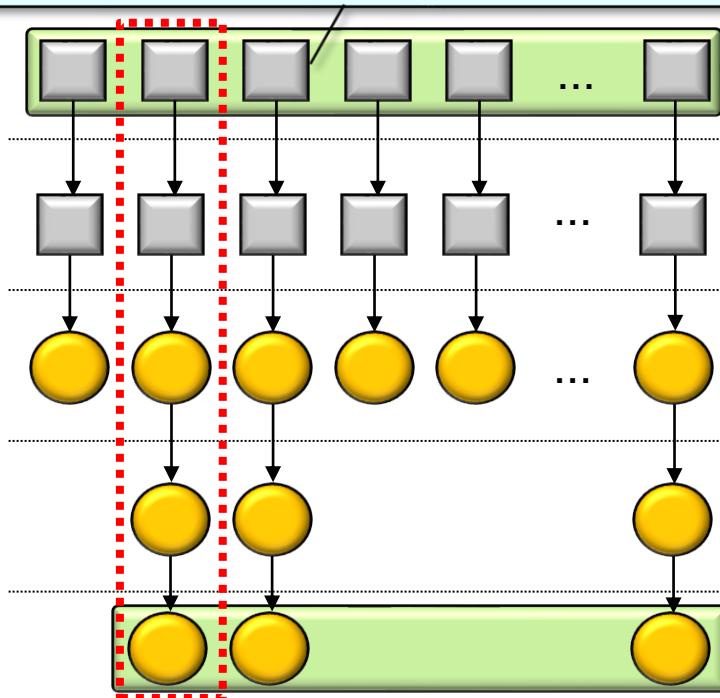
List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>

List  
<SearchResults>



This implementation is much more efficient & supports “short-circuit” operations

---

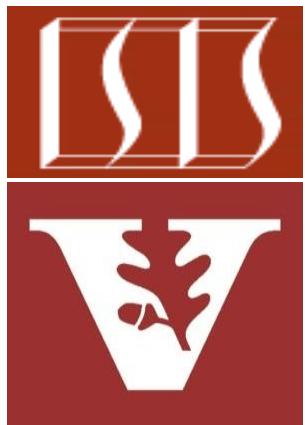
End of Visualize the Word  
Searcher.findWords() Method

# Recognize Common Java Streams Factory Methods

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)



Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



## Learning Objectives in this Part of the Lesson

---

- Recognize common factory methods used to create streams



---

# Common Factory Methods for Creating Streams

# Common Factory Methods for Creating Streams

---

- There are several common ways to obtain a stream



See [docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html)

# Common Factory Methods for Creating Streams

---

- There are several common ways to obtain a stream, e.g.
  - From a Java collection

```
List<String> wordsToFind =
 List.of("do", "re", "me", ...);
```

```
List<SearchResults> results =
 wordsToFind.stream()
 ...
```

or

```
List<SearchResults> results =
 wordsToFind.parallelStream()
 ...
```

# Common Factory Methods for Creating Streams

---

- There are several common ways to obtain a stream, e.g.
  - From a Java collection

```
List<String> wordsToFind =
 List.of("do", "re", "me", ...);
```

```
List<SearchResults> results =
 wordsToFind.stream()
```

...

or

```
List<SearchResults> results =
 wordsToFind.parallelStream()
```

...

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection

```
List<String> wordsToFind =
 List.of("do", "re", "me", ...);
```

```
List<SearchResults> results =
 wordsToFind.stream()
```

...

or

```
List<SearchResults> results =
 wordsToFind.parallelStream()
```

...

*We use this approach in the  
SimpleSearchStream program*

# Common Factory Methods for Creating Streams

---

- There are several common ways to obtain a stream, e.g.
  - From a Java collection

```
List<String> wordsToFind =
 List.of("do", "re", "me", ...);
```

```
List<SearchResults> results =
 wordsToFind.stream()
 ...
```

or

```
List<SearchResults> results =
 wordsToFind.parallelStream()
 ...
```

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection

```
List<String> wordsToFind =
 List.of("do", "re", "me", ...);
```

```
List<SearchResults> results =
 wordsToFind.stream()
```

...

or

```
List<SearchResults> results =
 wordsToFind.stream()
```

...

.parallel()

*A call to parallel() can appear anywhere in a stream & will have same effect as parallelStream()*

# Common Factory Methods for Creating Streams

---

- There are several common ways to obtain a stream, e.g.

- From a Java collection
- From an array

```
String[] a = {
 "a", "b", "c", "d", "e"
};
```

```
Stream<String> stream = Arrays.stream(a);
```

```
stream.forEach(s ->
 System.out.println(s));
```

or

```
stream.forEach(System.out::println);
```

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.

- From a Java collection
- From an array

```
String[] a = {
 "a", "b", "c", "d", "e"
};
```

```
Stream<String> stream = Arrays.stream(a);
```

```
stream.forEach(s ->
 System.out.println(s));
```

or

```
stream.forEach(System.out::println);
```

*Create stream containing  
all elements in an array*

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.

- From a Java collection
- From an array

```
String[] a = {
 "a", "b", "c", "d", "e"
};
```

```
Stream<String> stream = Arrays.stream(a);
```

```
stream.forEach(s ->
 System.out.println(s));
```

*Print all elements  
in the stream*

or

```
stream.forEach(System.out::println);
```

# Common Factory Methods for Creating Streams

---

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

```
String[] a = {
 "a", "b", "c", "d", "e"
};

Stream<String> stream = Stream.of(a);

stream.forEach(s ->
 System.out.println(s));
```

or

```
stream.forEach(System.out::println);
```

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

```
String[] a = {
 "a", "b", "c", "d", "e"
};

Stream<String> stream = Stream.of(a);

stream.forEach(s ->
 System.out.println(s));

or

stream.forEach(System.out::println);
```

*Create stream containing  
all elements in an array*

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.

- From a Java collection
- From an array
- From a static factory method

```
String[] a = {
 "a", "b", "c", "d", "e"
};

Stream<String> stream = Stream.of(a);
```

```
stream.forEach(s ->
 System.out.println(s));
```

*Print all elements  
in the stream*

or

```
stream.forEach(System.out::println);
```

# Common Factory Methods for Creating Streams

---

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

```
Stream.iterate(new BigInteger[] {BigInteger.ONE,
 BigInteger.ONE},
 f -> new BigInteger[] {f[1],
 f[0].add(f[1])})
 .map(f -> f[0])
 .limit(100)
 .forEach(System.out::println);
```

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

*Generate & print the first 100 Fibonacci #'s*

```
Stream.iterate(new BigInteger[]{BigInteger.ONE,
 BigInteger.ONE},
 f -> new BigInteger[]{f[1],
 f[0].add(f[1])})
.map(f -> f[0])
.limit(100)
.forEach(System.out::println);
```

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

*Create the "seed," which defines the initial element in the stream*

```
Stream.iterate(new BigInteger[] {BigInteger.ONE,
 BigInteger.ONE} ,
 f -> new BigInteger[] {f[1],
 f[0].add(f[1]) })
.map(f -> f[0])
.limit(100)
.forEach(System.out::println);
```

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

```
Stream.iterate(new BigInteger[] {BigInteger.ONE,
 BigInteger.ONE} ,
 f -> new BigInteger[] {f[1],
 f[0].add(f[1]) })

.map(f -> f[0])
.limit(100)
.forEach(System.out::println);
```

A lambda function applied  
to the previous element to  
produce a new element

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

```
Stream.iterate(new BigInteger[] {BigInteger.ONE,
 BigInteger.ONE} ,
 f -> new BigInteger[] {f[1],
 f[0].add(f[1]) })

.map(f -> f[0])
.limit(100)
.forEach(System.out::println);
```

*Convert the array to its first element*

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

```
Stream.iterate(new BigInteger[] {BigInteger.ONE,
 BigInteger.ONE} ,
 f -> new BigInteger[] {f[1],
 f[0].add(f[1]) })
.map(f -> f[0])
.limit(100) ——————
.forEach(System.out::println);
```

*Short-circuit operation limits  
the stream to 100 elements*

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

```
Stream.iterate(new BigInteger[]{BigInteger.ONE,
 BigInteger.ONE},
 f -> new BigInteger[]{f[1],
 f[0].add(f[1])})
.map(f -> f[0])
.limit(100)
.forEach(System.out::println);
```

*Print the first 100  
Fibonacci #'s*

# Common Factory Methods for Creating Streams

- There are several common ways to obtain a stream, e.g.
  - From a Java collection
  - From an array
  - From a static factory method

```
Stream.iterate(new BigInteger[]{BigInteger.ONE,
 BigInteger.ONE},
 f -> new BigInteger[]{f[1],
 f[0].add(f[1])})

• .parallel()
• .map(f -> f[0])
• .limit(100)
• .forEach(System.out::println);
```

*Avoid using iterate() in a parallel stream!*

---

# End of Recognize Common Java Streams Factory Methods

# **Understand Java Streams**

## **Aggregate Operations**

**Douglas C. Schmidt**

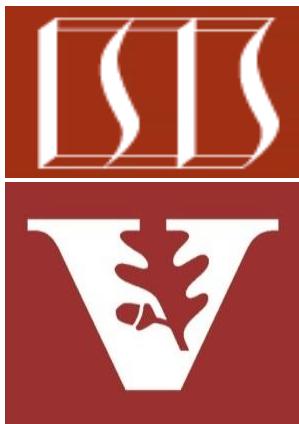
**d.schmidt@vanderbilt.edu**

**[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

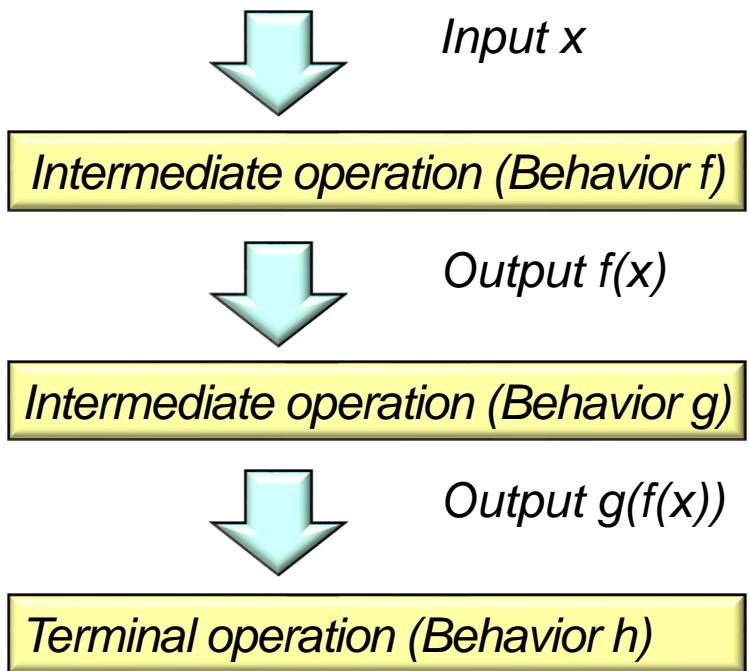
**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

---

- Understand the structure & functionality of stream aggregate operations



---

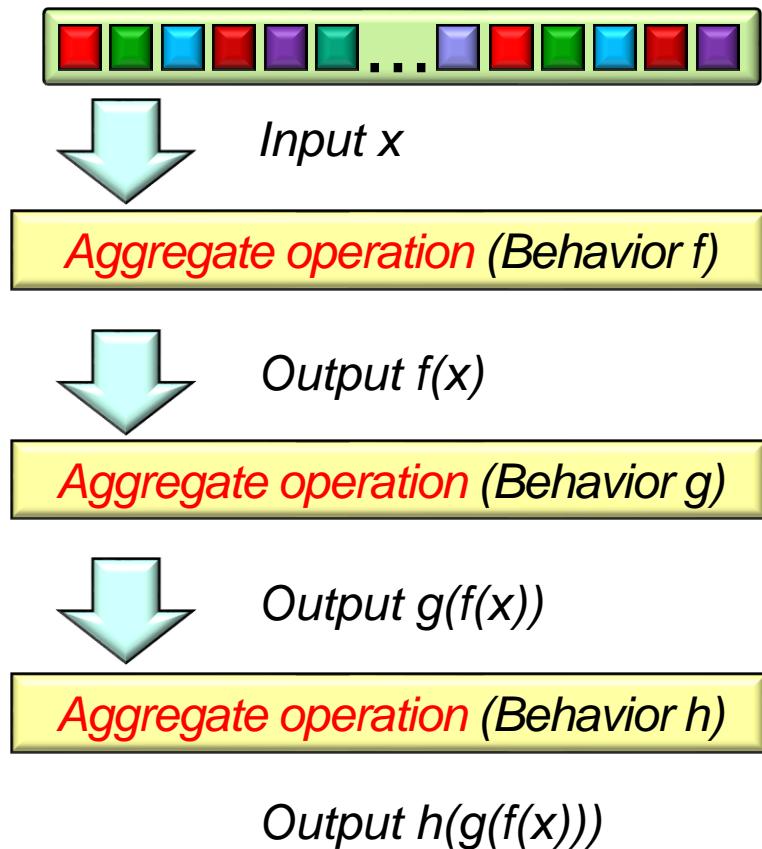
# Overview of Stream Aggregate Operations

# Overview of Stream Aggregate Operations

- An aggregate operation is a higher-order function that applies a “behavior” on elements in a stream



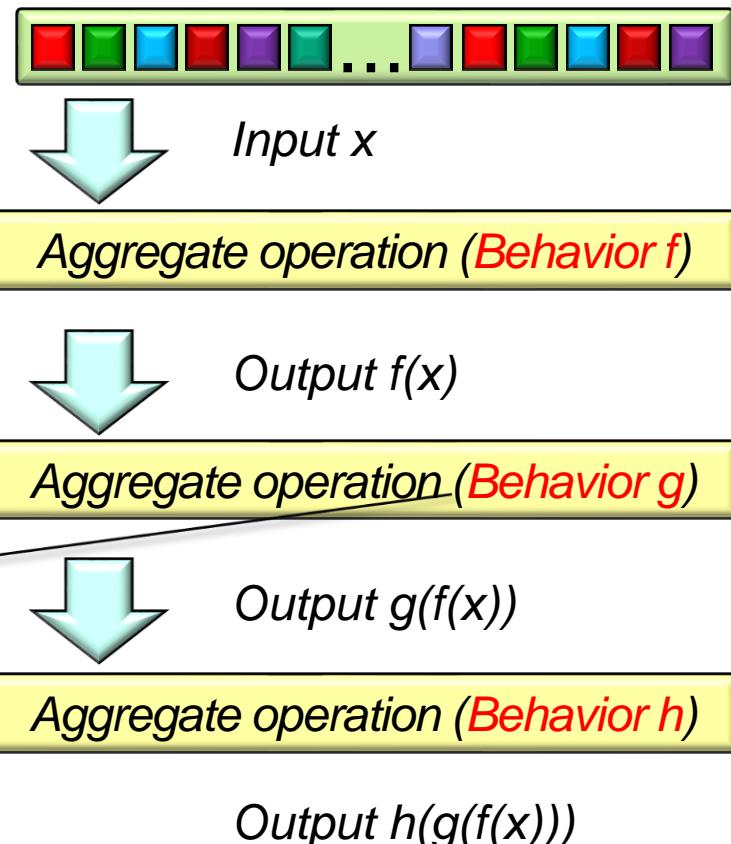
A “higher order function” is a function that is passed a function as a param



See [en.wikipedia.org/wiki/Higher-order\\_function](https://en.wikipedia.org/wiki/Higher-order_function)

# Overview of Stream Aggregate Operations

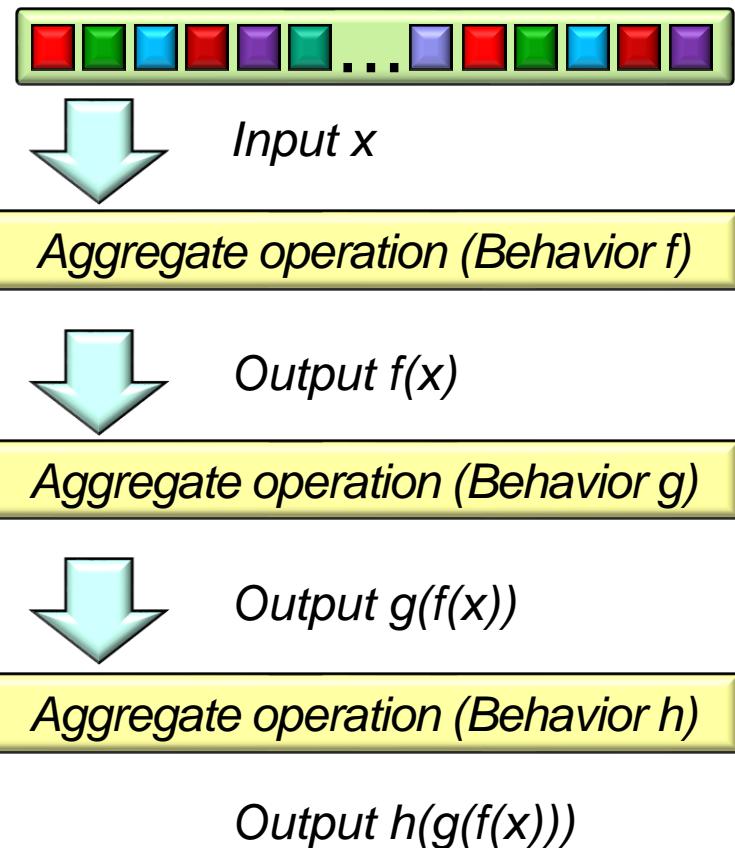
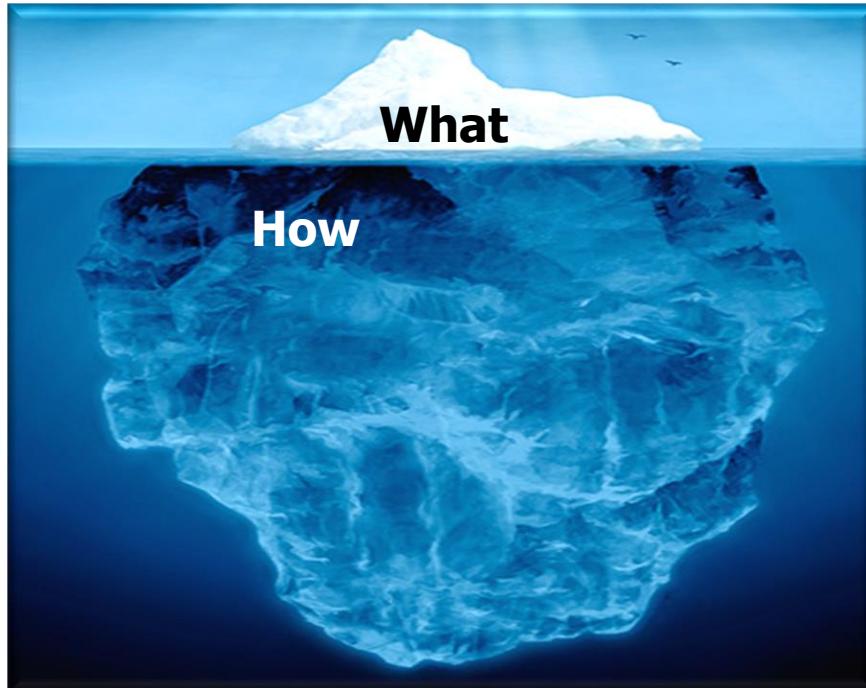
- An aggregate operation is a higher-order function that applies a “behavior” on elements in a stream



*The behavior can be a lambda or method reference to a function, predicate, consumer, supplier, etc.*

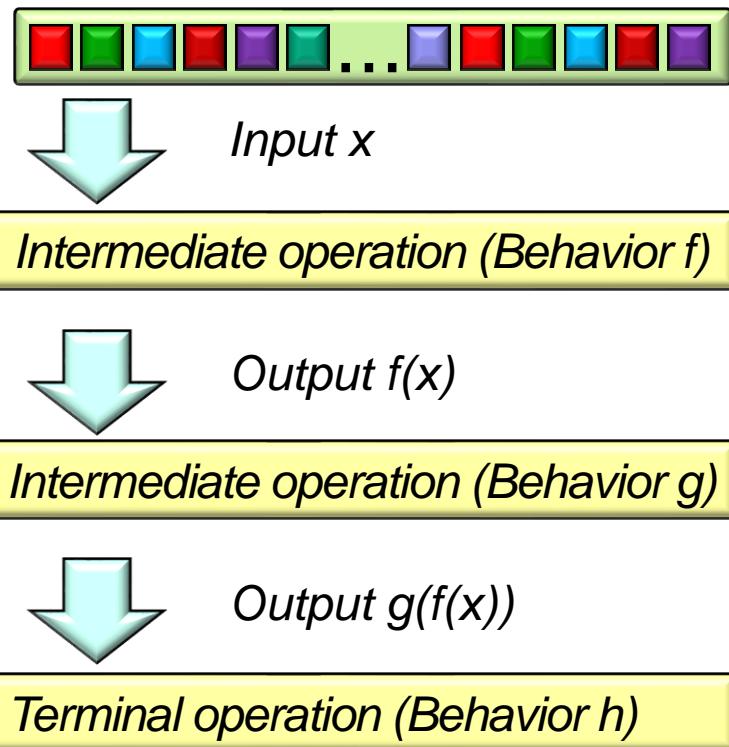
# Overview of Stream Aggregate Operations

- Aggregate operations form a declarative pipeline that emphasizes the “what” & deemphasizes the “how”



# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

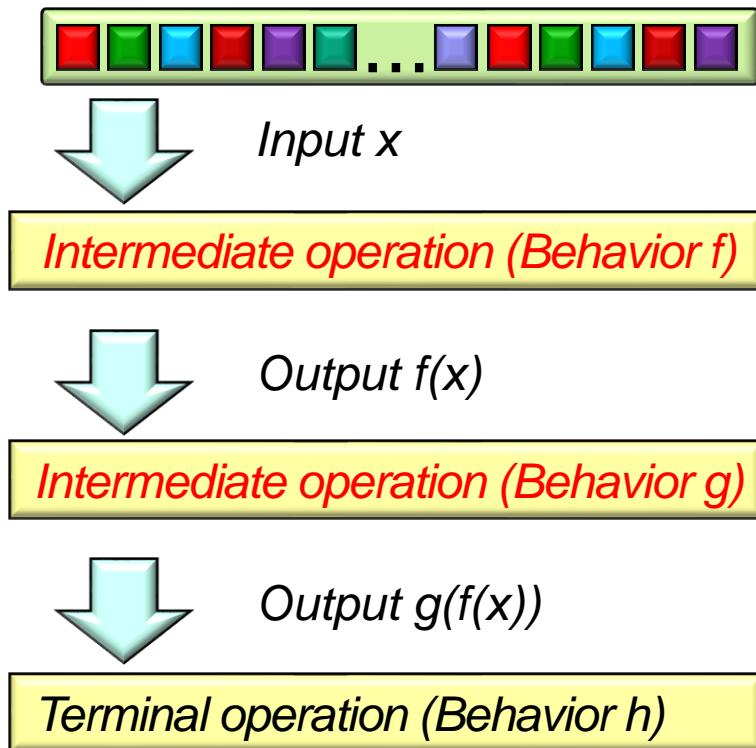


# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- Process elements in their input stream & yield an output stream
  - e.g., filter(), map(), flatMap(), takeWhile(), dropWhile(), etc.



# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- Intermediate operations**

- Process elements in their input stream & yield an output stream
  - e.g., filter(), map(), flatMap(), takeWhile(), dropWhile(), etc.

*Intermediate operations are optional.*

```
long hamletCharacters = Stream
 .of("horatio", "laertes",
 "Hamlet", ...)
 .count();
```



*Input x*

**OPTIONAL**

*Terminal operation (Behavior h)*

# Overview of Stream Aggregate Operations

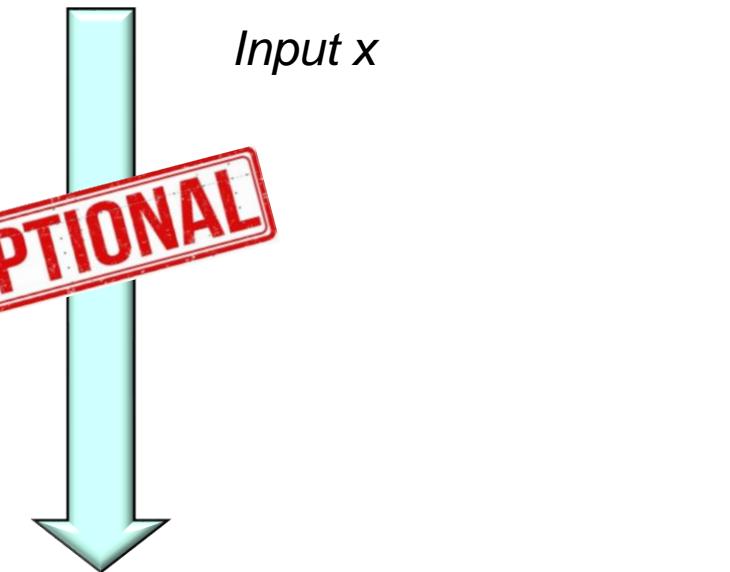
- There are two types of aggregate operations

- Intermediate operations**

- Process elements in their input stream & yield an output stream
  - e.g., filter(), map(), flatMap(), takeWhile(), dropWhile(), etc.

*The semantics of count() are now weird..*

```
long hamletCharacters = Stream
 .of("horatio", "laertes",
 "Hamlet", ...)
 .peek(System.out::print)
 .count();
```



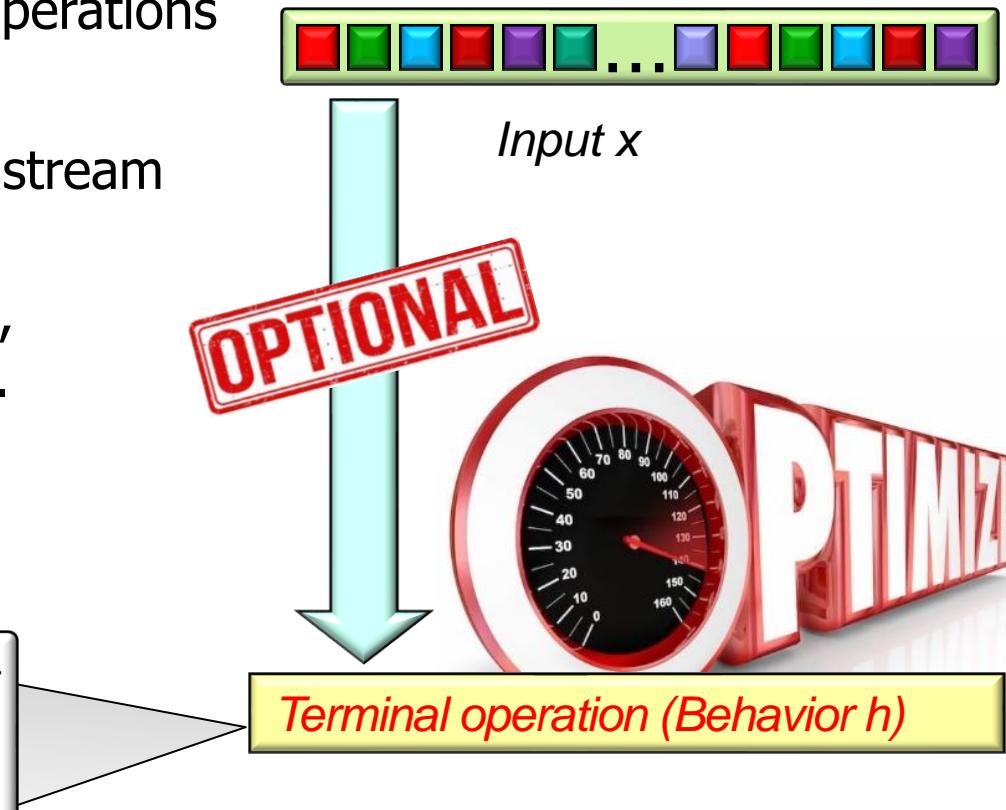
# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- Process elements in their input stream & yield an output stream
  - e.g., filter(), map(), flatMap(), takeWhile(), dropWhile(), etc.

```
long hamletCharacters = Stream
 .of("horatio", "laertes",
 "Hamlet", ...)
 .count();
```



Newer versions of Java optimize streams containing no intermediate operations

# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- Process elements in their input stream & yield an output stream
- Intermediate operations can be further classified via several dimensions

|           | Run-to-completion                | Short-Circuiting                        |
|-----------|----------------------------------|-----------------------------------------|
| Stateful  | distinct(), skip(), sorted()     | limit(), takeWhile(), dropWhile(), etc. |
| Stateless | filter(), map(), flatMap(), etc. | N/A                                     |

# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- Process elements in their input stream & yield an output stream
- Intermediate operations can be further classified via several dimensions, e.g.
  - Stateful
    - Store info from a prior invocation for use in a future invocation

|           | Run-to-completion                                                                   | Short-Circuiting                                                                           |
|-----------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| Stateful  | <code>distinct()</code> ,<br><code>skip()</code> ,<br><code>sorted()</code>         | <code>limit()</code> ,<br><code>takeWhile()</code> ,<br><code>dropWhile()</code> ,<br>etc. |
| Stateless | <code>filter()</code> ,<br><code>map()</code> ,<br><code>flatMap()</code> ,<br>etc. | N/A                                                                                        |



# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- Process elements in their input stream & yield an output stream
- Intermediate operations can be further classified via several dimensions, e.g.
  - Stateful
  - Stateless
    - Do not store info from any prior invocations

|           | Run-to-completion                | Short-Circuiting                        |
|-----------|----------------------------------|-----------------------------------------|
| Stateful  | distinct(), skip(), sorted()     | limit(), takeWhile(), dropWhile(), etc. |
| Stateless | filter(), map(), flatMap(), etc. | N/A                                     |



See [javapapers.com/java/java-stream-api](http://javapapers.com/java/java-stream-api)

# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- Process elements in their input stream & yield an output stream
- Intermediate operations can be further classified via several dimensions, e.g.
  - Stateful
  - Stateless
    - Do not store info from any prior invocations

|           | Run-to-completion                | Short-Circuiting                        |
|-----------|----------------------------------|-----------------------------------------|
| Stateful  | distinct(), skip(), sorted()     | limit(), takeWhile(), dropWhile(), etc. |
| Stateless | filter(), map(), flatMap(), etc. | N/A                                     |

*Stateless operations often require significantly fewer processing & memory resources than stateful operations!*

# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- Process elements in their input stream & yield an output stream
- Intermediate operations can be further classified via several dimensions, e.g.
  - Stateful
  - Stateless
  - Run-to-completion
    - Process all elements in the input stream

|           | Run-to-completion                                                                   | Short-Circuiting                                                                           |
|-----------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| Stateful  | <code>distinct()</code> ,<br><code>skip()</code> ,<br><code>sorted()</code>         | <code>limit()</code> ,<br><code>takeWhile()</code> ,<br><code>dropWhile()</code> ,<br>etc. |
| Stateless | <code>filter()</code> ,<br><code>map()</code> ,<br><code>flatMap()</code> ,<br>etc. | N/A                                                                                        |



See [en.wikipedia.org/wiki/Run\\_to\\_completion\\_scheduling](https://en.wikipedia.org/wiki/Run_to_completion_scheduling)

# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- Intermediate operations**

- Process elements in their input stream & yield an output stream

- Intermediate operations can be further classified via several dimensions, e.g.

- Stateful
- Stateless
- Run-to-completion
- Short-circuiting
  - Make stream operate on a reduced size

|           | Run-to-completion                | Short-Circuiting                        |
|-----------|----------------------------------|-----------------------------------------|
| Stateful  | distinct(), skip(), sorted()     | limit(), takeWhile(), dropWhile(), etc. |
| Stateless | filter(), map(), flatMap(), etc. | N/A                                     |

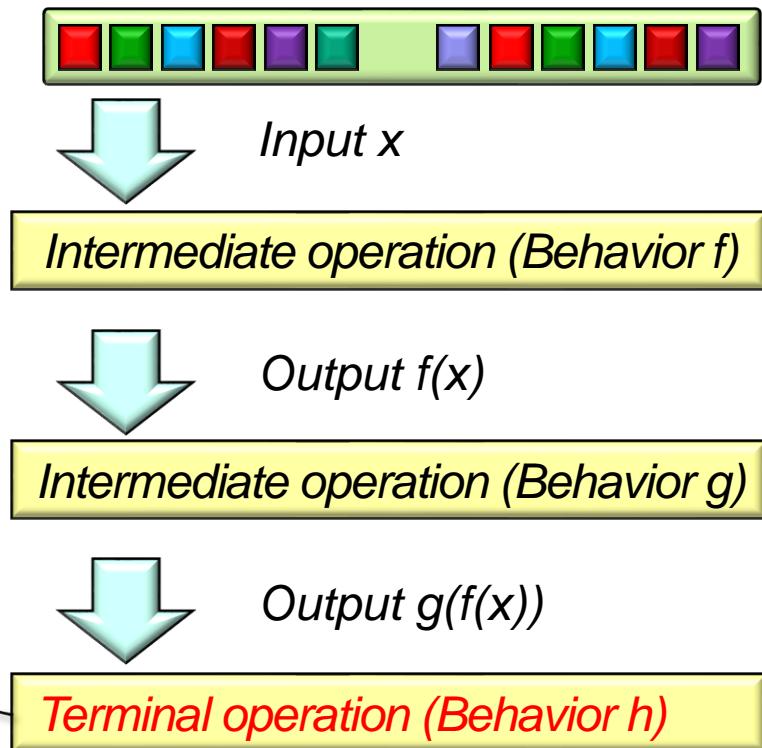


# Overview of Stream Aggregate Operations

- There are two types of aggregate operations
  - **Intermediate operations**
  - **Terminal operations**
    - Trigger intermediate operations & produce a non-stream result
      - e.g., `forEach()`, `reduce()`, `collect()`, `findAny()`, etc.



*A stream must have one (& only one) terminal operation*



# Overview of Stream Aggregate Operations

---

- There are two types of aggregate operations

- **Intermediate operations**

- **Terminal operations**

- Trigger intermediate operations & produce a non-stream result
- Terminal operations can also be classified via several dimensions

| Operation Type    | Examples                                                    |
|-------------------|-------------------------------------------------------------|
| Run-to-completion | reduce(), collect(), forEach(), etc.                        |
| Short-circuiting  | allMatch(), anyMatch(), findAny(), findFirst(), noneMatch() |

# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- **Terminal operations**

- Trigger intermediate operations & produce a non-stream result
- Terminal operations can also be classified via several dimensions, e.g.
  - Run-to-completion
    - Terminate only after processing all elements in the stream

| Operation Type    | Examples                                                    |
|-------------------|-------------------------------------------------------------|
| Run-to-completion | reduce(), collect(), forEach(), etc.                        |
| Short-circuiting  | allMatch(), anyMatch(), findAny(), findFirst(), noneMatch() |



# Overview of Stream Aggregate Operations

- There are two types of aggregate operations

- **Intermediate operations**

- **Terminal operations**

- Trigger intermediate operations & produce a non-stream result

- Terminal operations can also be classified via several dimensions, e.g.

- Run-to-completion

- Short-circuiting

- May cause a stream to terminate before processing all values

| Operation Type    | Examples                                                    |
|-------------------|-------------------------------------------------------------|
| Run-to-completion | reduce(), collect(), forEach(), etc.                        |
| Short-circuiting  | allMatch(), anyMatch(), findAny(), findFirst(), noneMatch() |



---

# End of Understand Java Streams Aggregate Operations

# **Understand Java Streams Short-Circuit Aggregate Operations**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)**

**Professor of Computer Science**

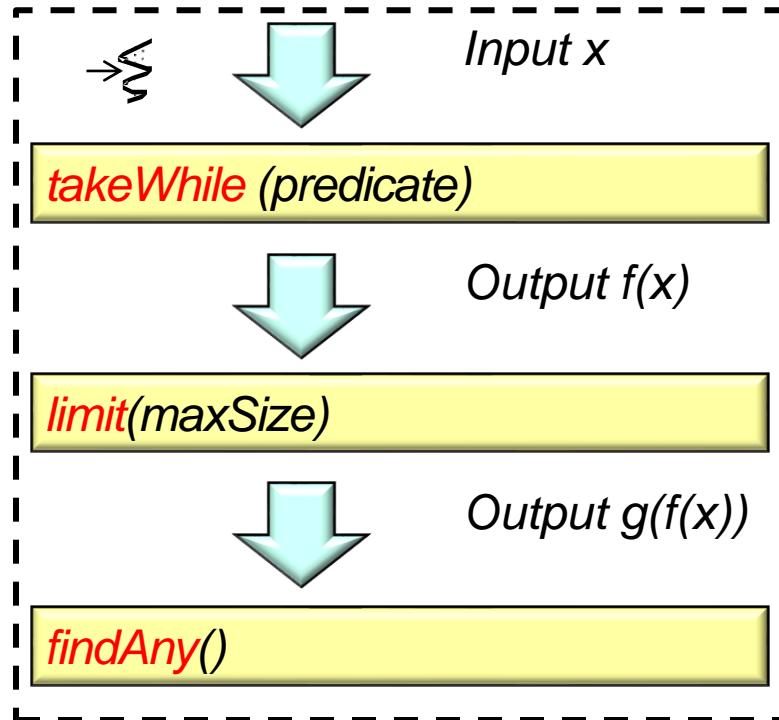
**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of stream aggregate operations
- Understand the Java stream “short-circuit” aggregate operations

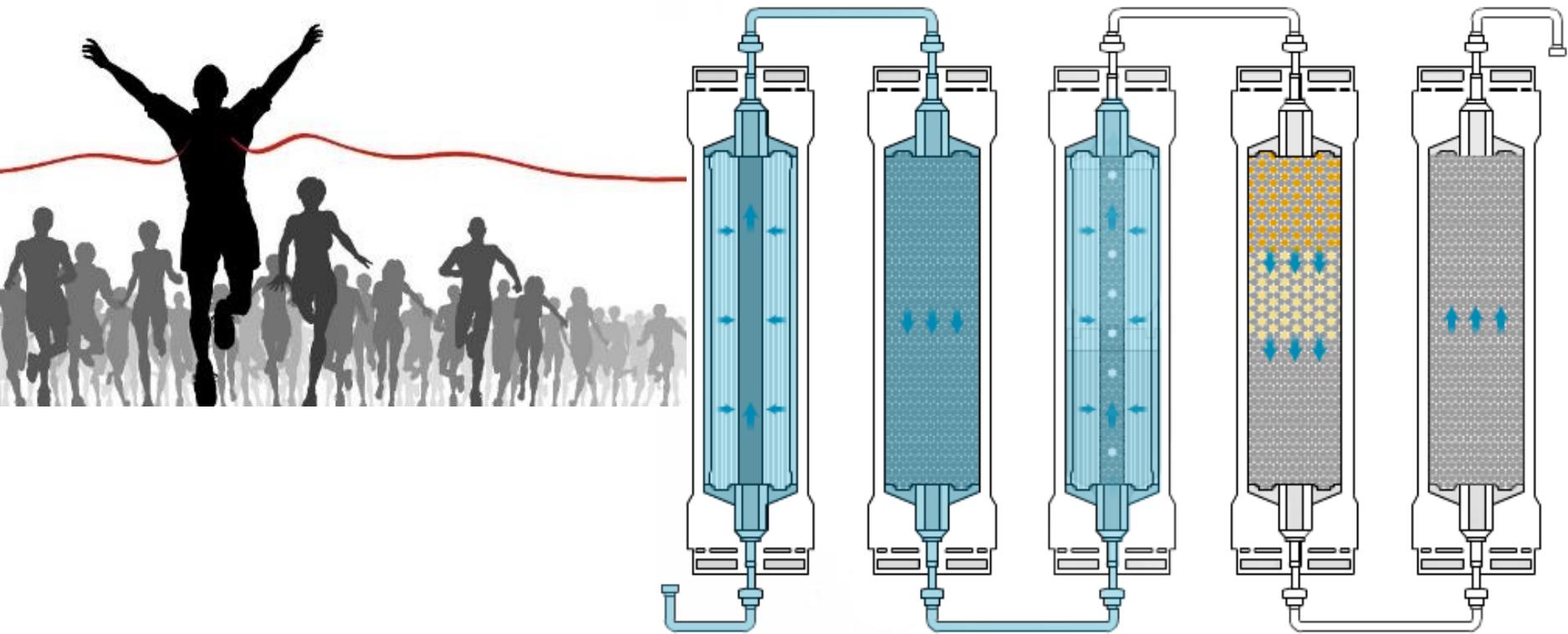


---

# Java Streams Short-Circuit Operations

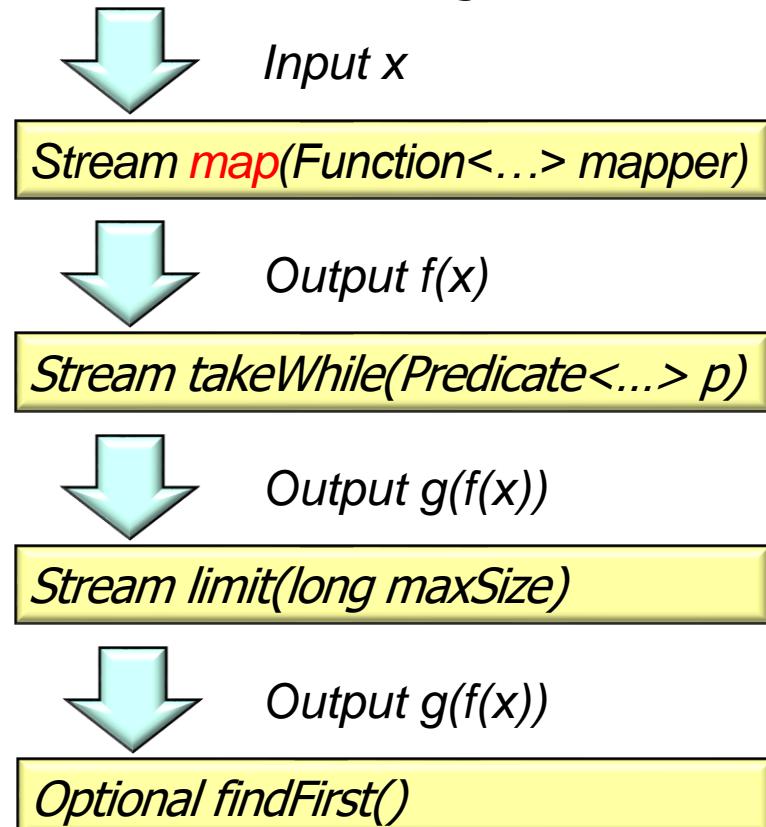
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream



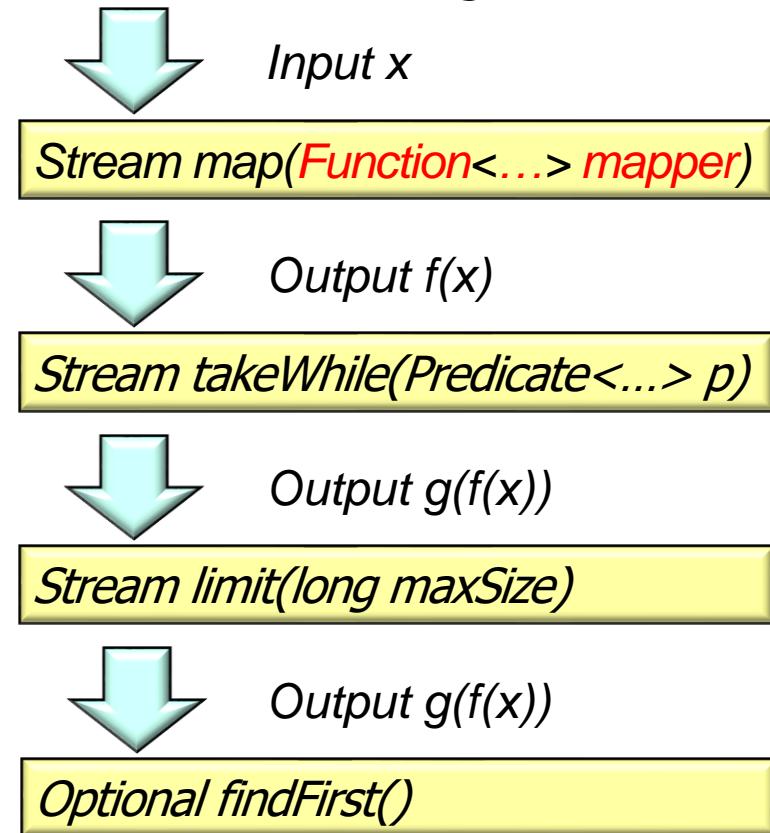
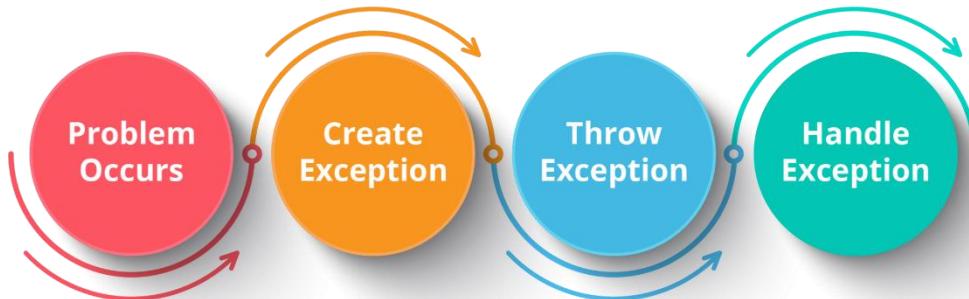
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - map() processes all of the elements in its input stream



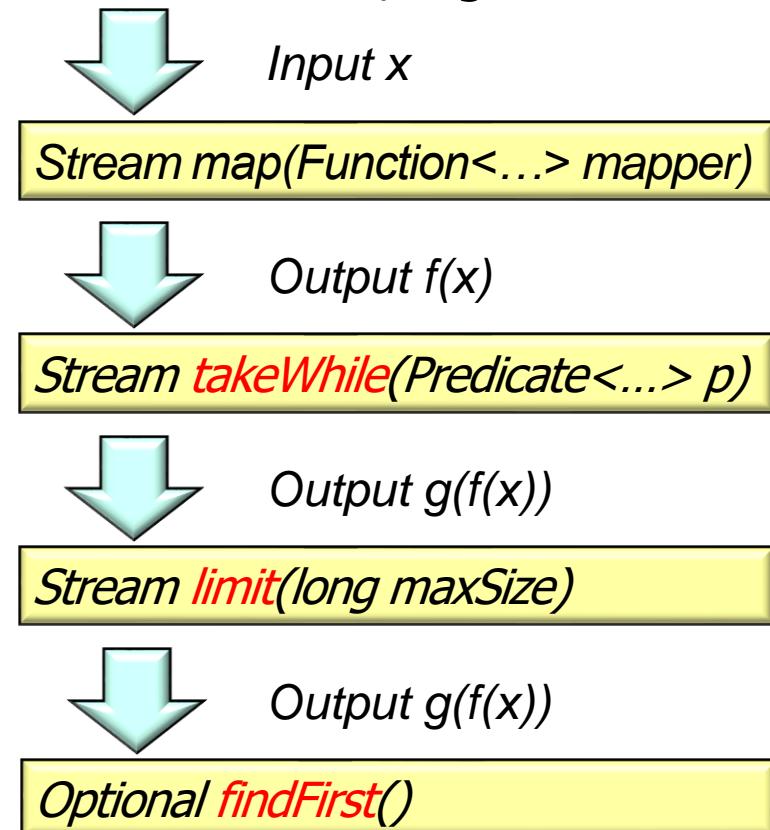
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - map() processes all of the elements in its input stream
  - Unless a behavior throws an exception..



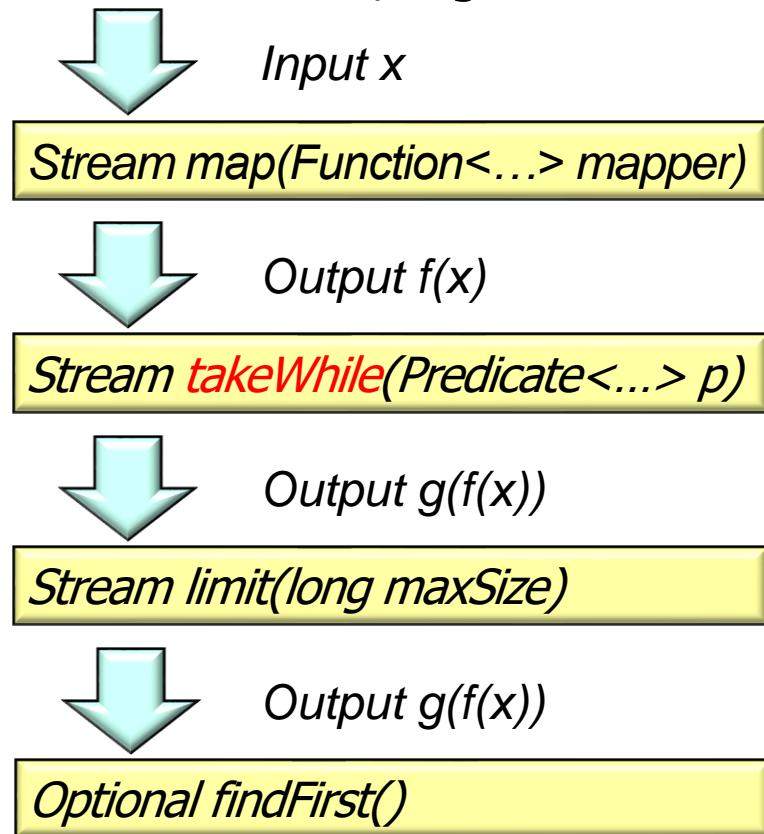
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - `map()` processes all of the elements in its input stream
  - “Short-circuit” operations halt further processing after reaching their condition



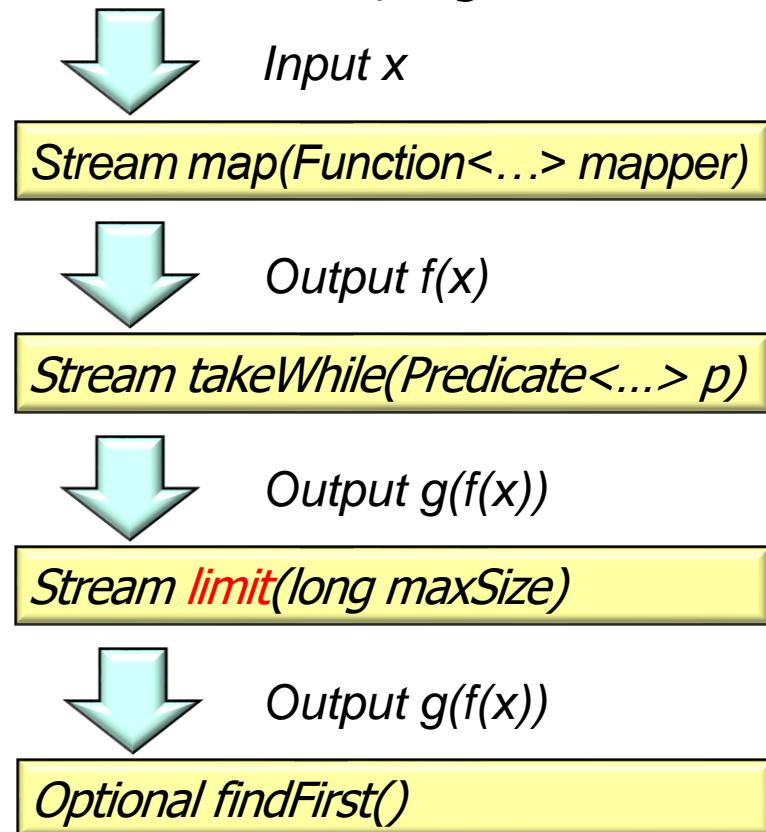
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - map() processes all of the elements in its input stream
  - “Short-circuit” operations halt further processing after reaching their condition
    - takeWhile()
      - A short-circuit intermediate operation that returns a stream consisting of a subset of elements taken from this stream that match the given predicate



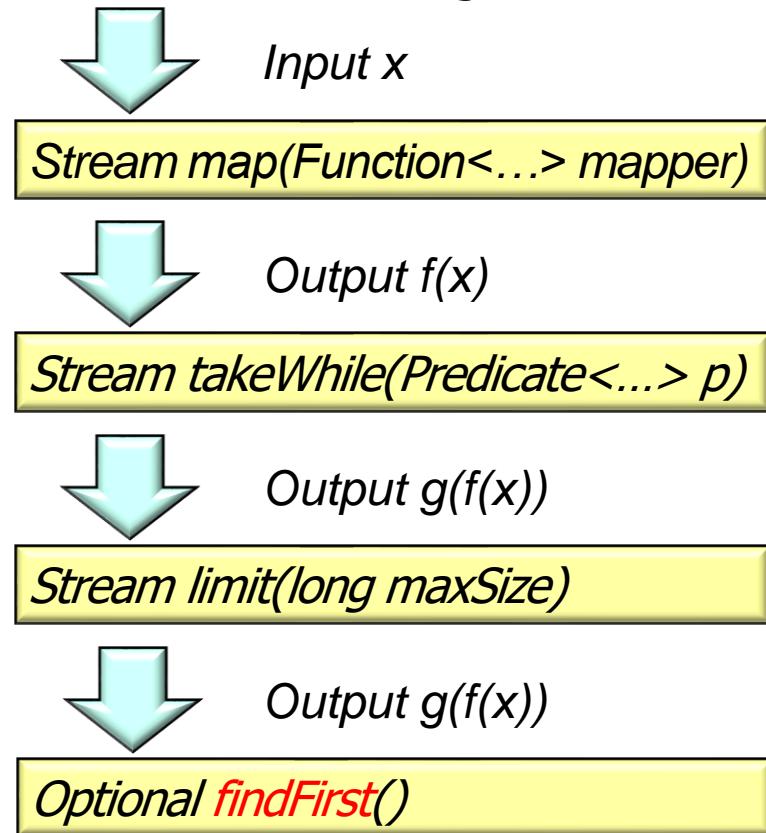
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - map() processes all of the elements in its input stream
  - “Short-circuit” operations halt further processing after reaching their condition
    - takeWhile()
    - limit()
      - A short-circuit intermediate operation that causes a stream to operate on a reduced size



# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - map() processes all of the elements in its input stream
  - “Short-circuit” operations halt further processing after reaching their condition
    - takeWhile()
    - limit()
  - findFirst(), findAny(), anyMatch(), allMatch(), & noneMatch()
    - Short-circuit terminal operations can finish before traversing all elements in the underlying stream



---

# End of Understand Java Streams Short-Circuit Aggregate Operations

# **Understand Java Streams Short-Circuit Aggregate Operations**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)**

**Professor of Computer Science**

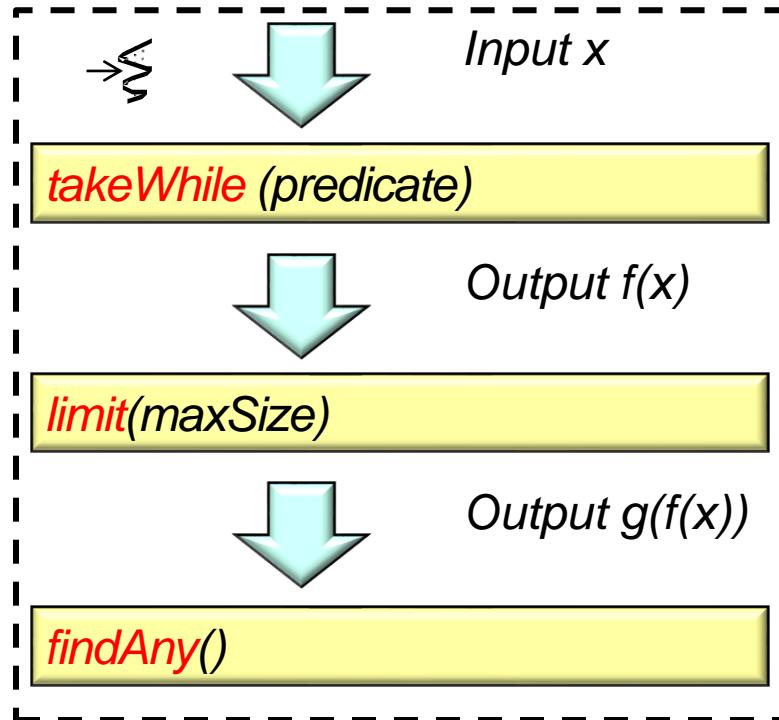
**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of stream aggregate operations
- Understand the Java stream “short-circuit” aggregate operations

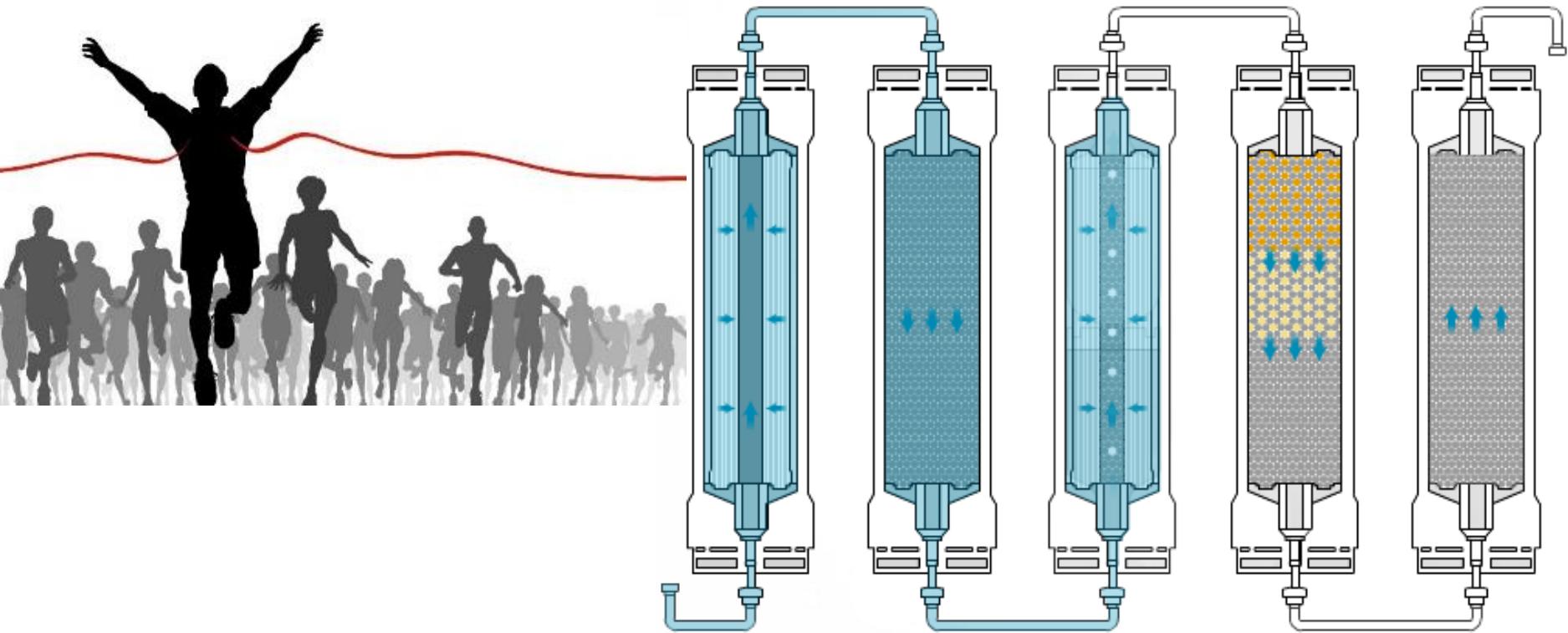


---

# Java Streams Short-Circuit Operations

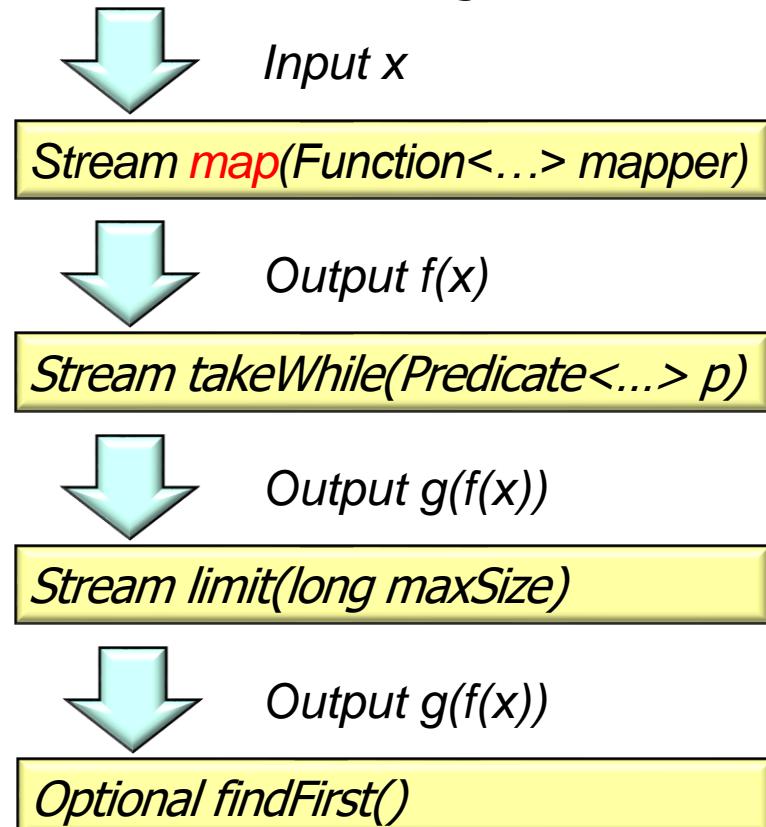
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream



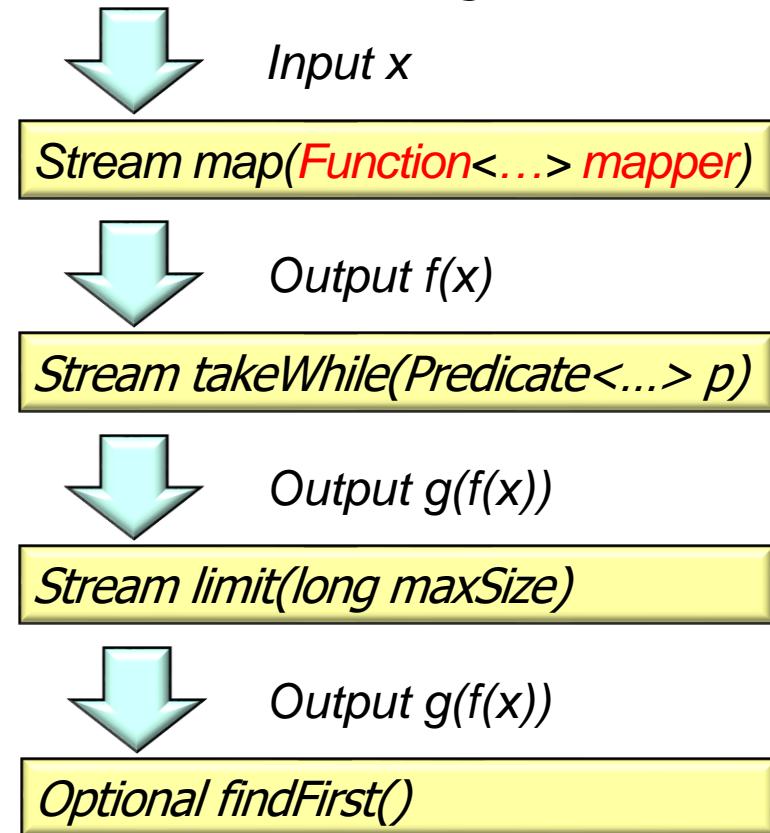
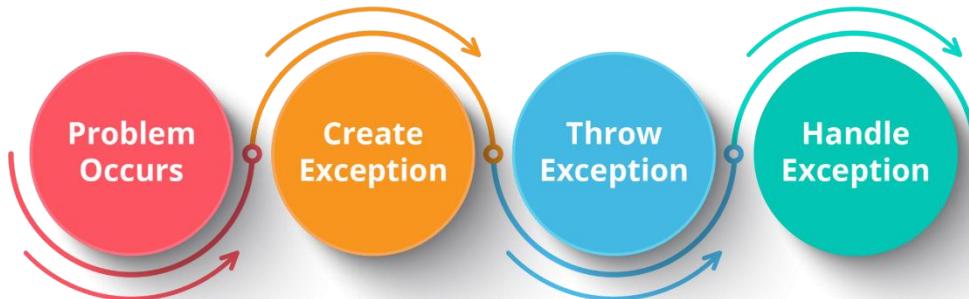
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - map() processes all of the elements in its input stream



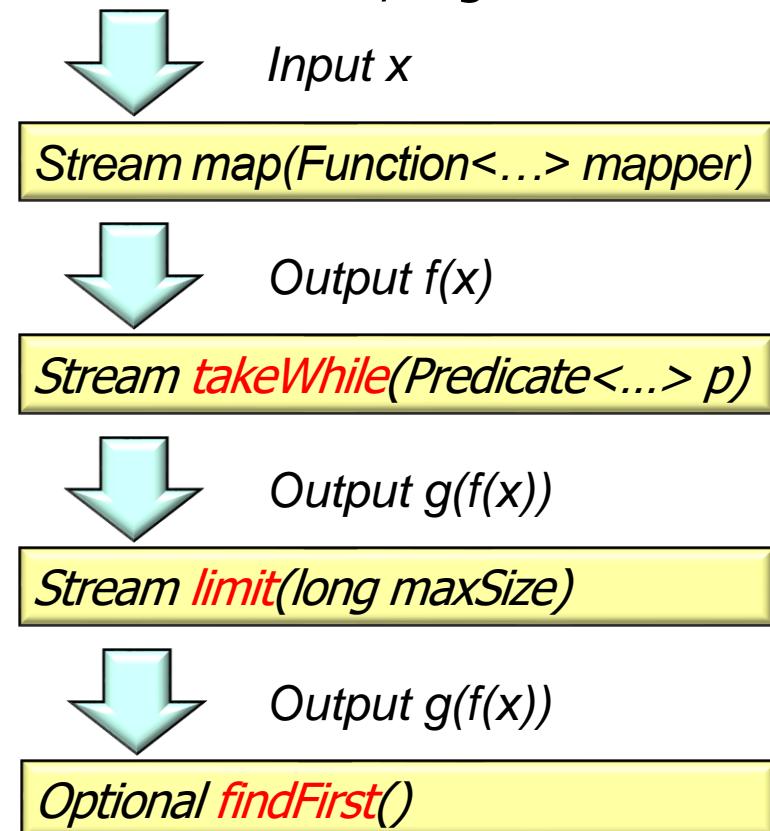
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - map() processes all of the elements in its input stream
  - Unless a behavior throws an exception..



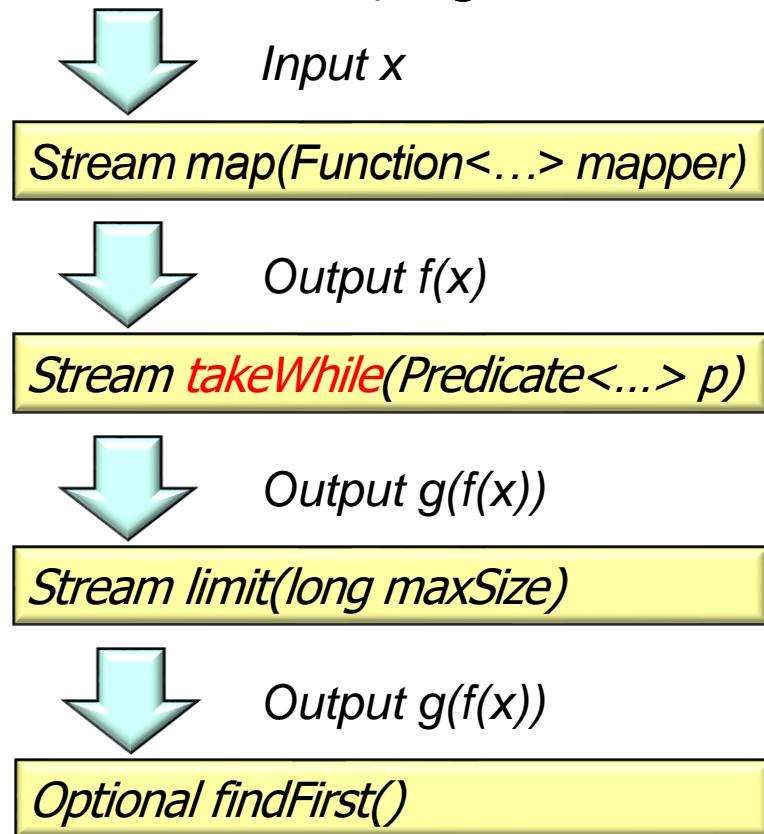
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - `map()` processes all of the elements in its input stream
  - “Short-circuit” operations halt further processing after reaching their condition



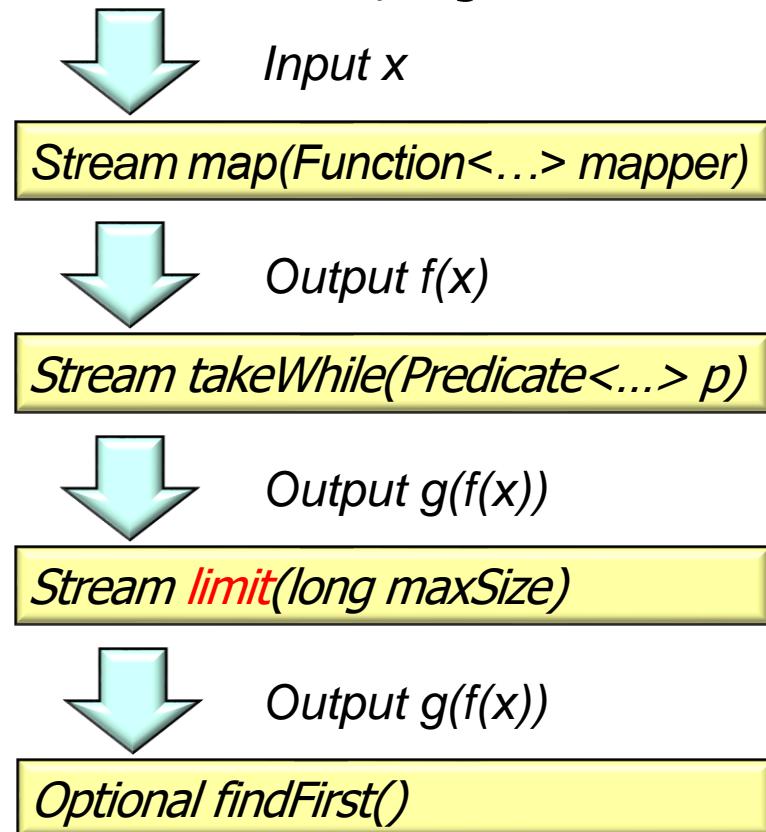
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - `map()` processes all of the elements in its input stream
  - “Short-circuit” operations halt further processing after reaching their condition
    - `takeWhile()`
      - A short-circuit intermediate operation that returns a stream consisting of a subset of elements taken from this stream that match the given predicate



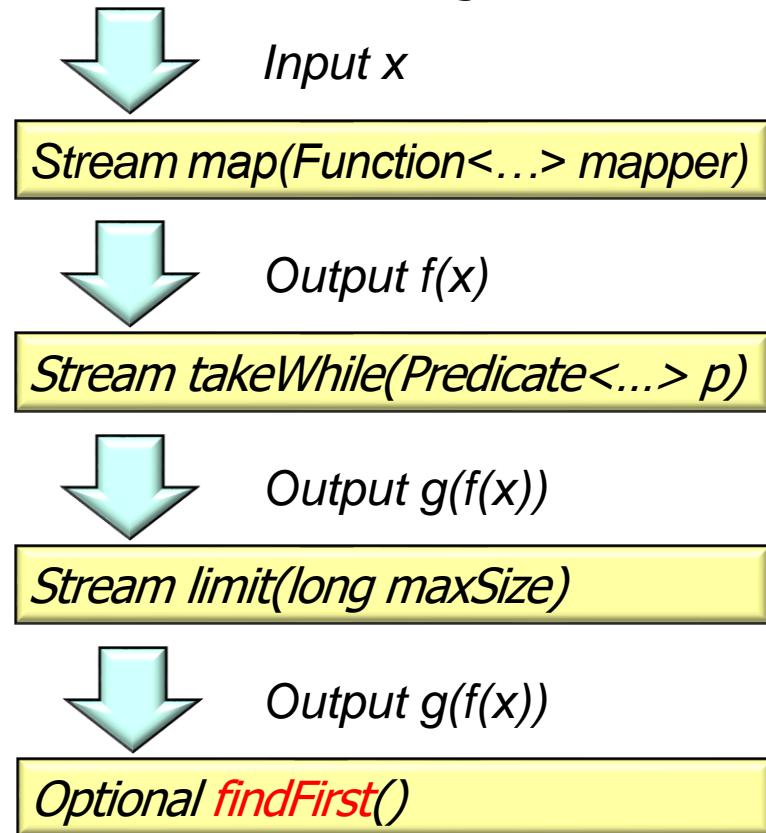
# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - map() processes all of the elements in its input stream
  - “Short-circuit” operations halt further processing after reaching their condition
    - takeWhile()
    - limit()
      - A short-circuit intermediate operation that causes a stream to operate on a reduced size



# Java Streams Short-Circuit Operations

- An aggregate operation *may* process all elements in a stream, e.g.
  - map() processes all of the elements in its input stream
  - “Short-circuit” operations halt further processing after reaching their condition
    - takeWhile()
    - limit()
  - findFirst(), findAny(), anyMatch(), allMatch(), & noneMatch()
    - Short-circuit terminal operations can finish before traversing all elements in the underlying stream



---

# End of Understand Java Streams Short-Circuit Aggregate Operations

# **Understand Java Streams Intermediate Operations map() & mapToInt()**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

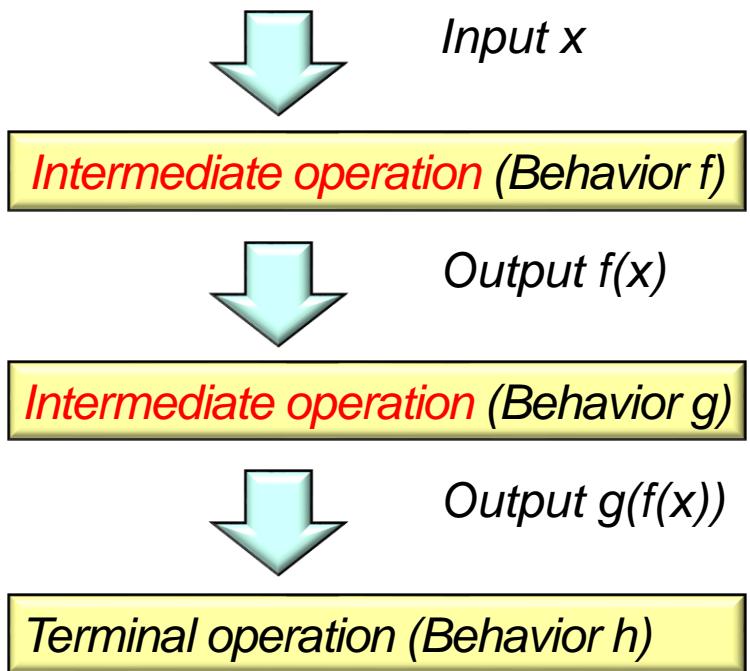
**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

---

- Understand the structure & functionality of stream aggregate operations
  - Intermediate operations

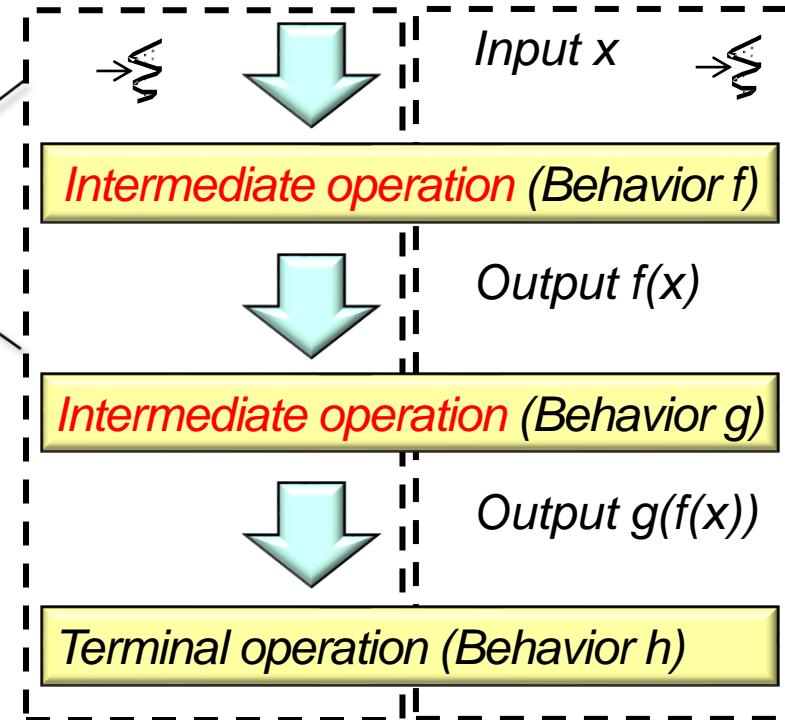


# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of stream aggregate operations
  - Intermediate operations



*These operations apply to both sequential & parallel streams*



Being a good streams programmer makes you a better parallel streams programmer

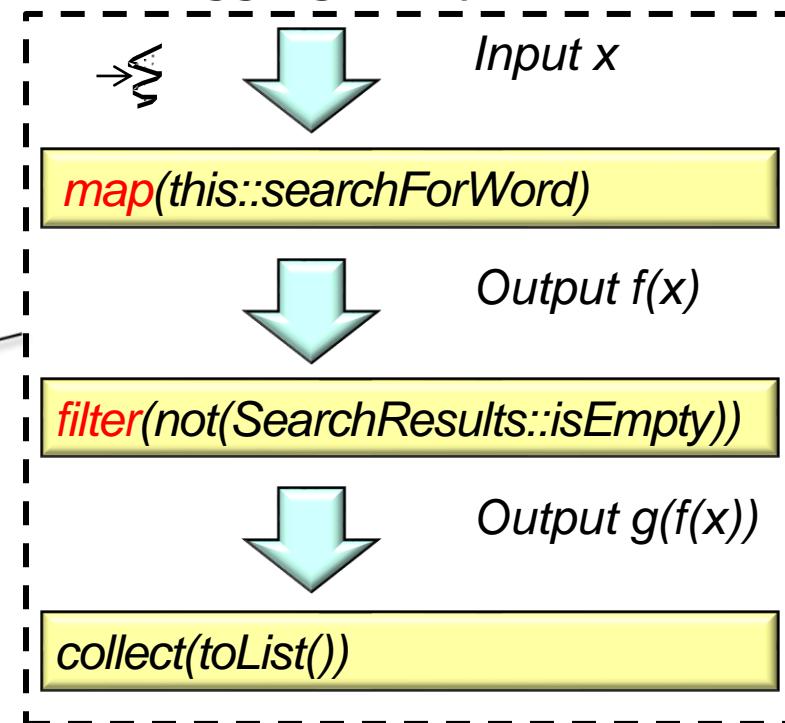
# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of stream aggregate operations
  - Intermediate operations

**Input String to Search**  
Let's start at the very beginning..

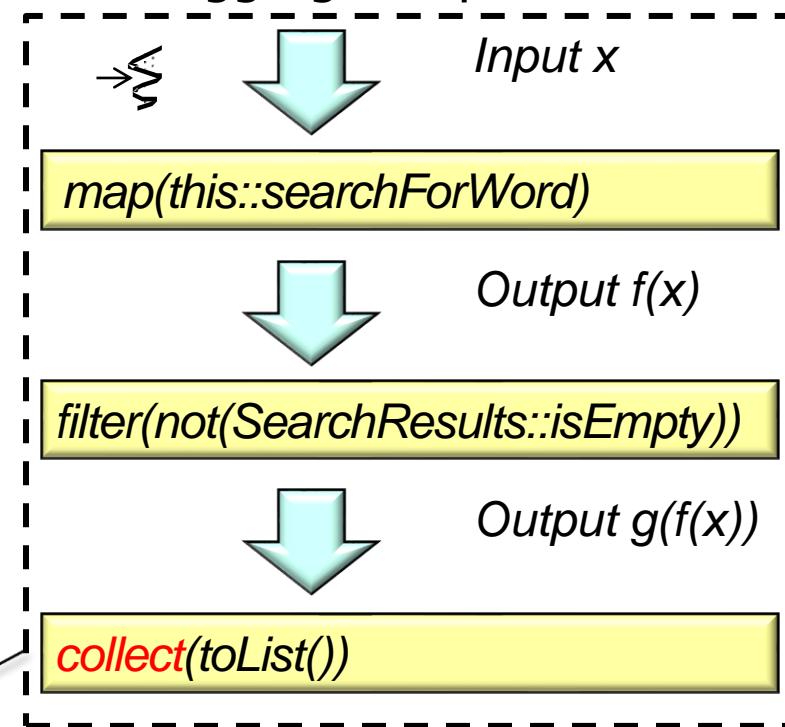
**Search Words**  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

*We continue to showcase the SimpleSearchStream program*



# Learning Objectives in this Part of the Lesson

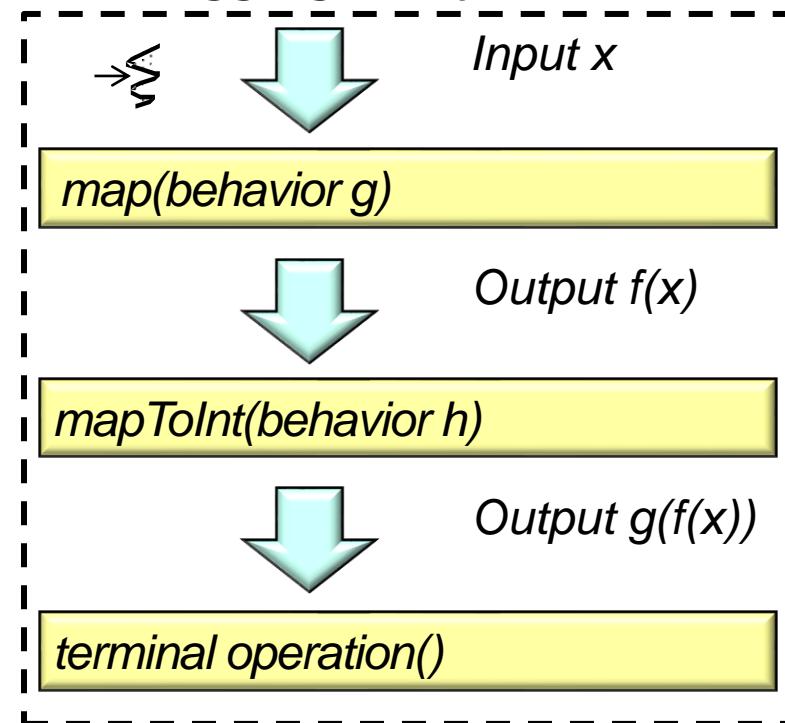
- Understand the structure & functionality of stream aggregate operations
  - Intermediate operations



*Intermediate operations are "lazy" & run only after terminal operator is reached.*

# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of stream aggregate operations
  - Intermediate operations
  - `map()` & `mapToInt()`



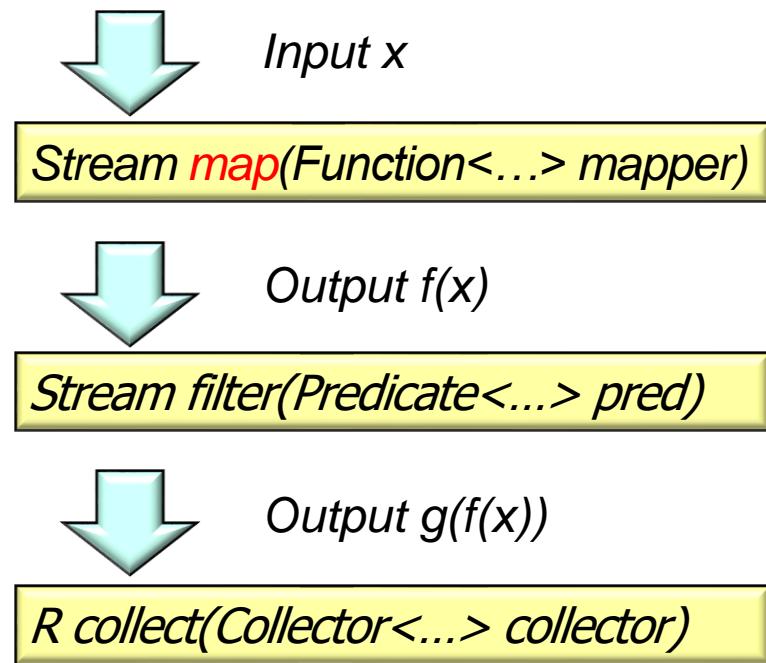
These are both stateless, run-to-completion operations

---

# Overview of the map() Intermediate Operation

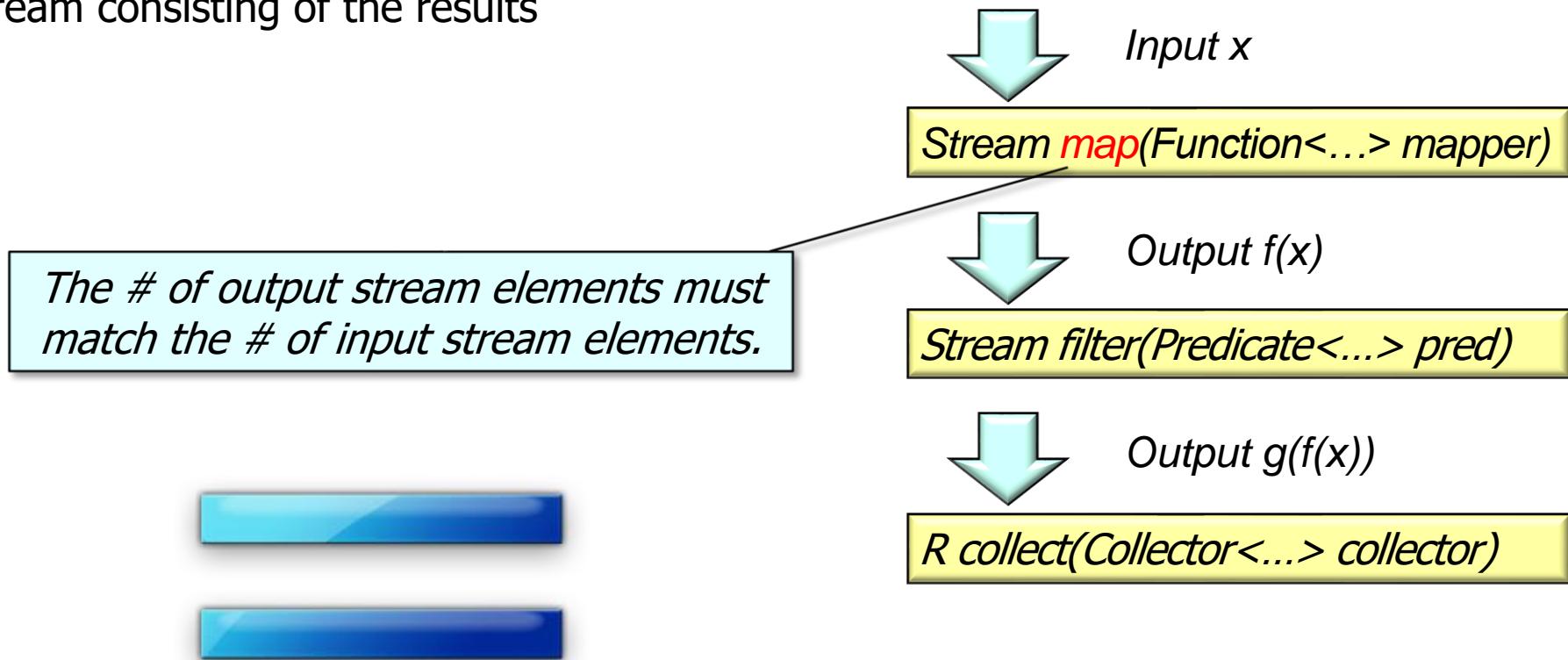
# Overview of the map() Intermediate Operation

- Applies a mapper function to every element of the input stream & returns an output stream consisting of the results



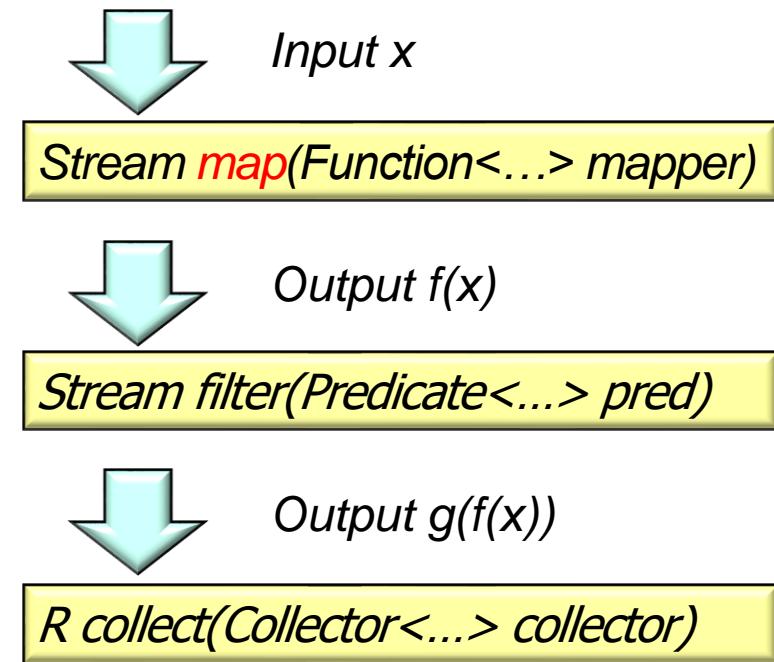
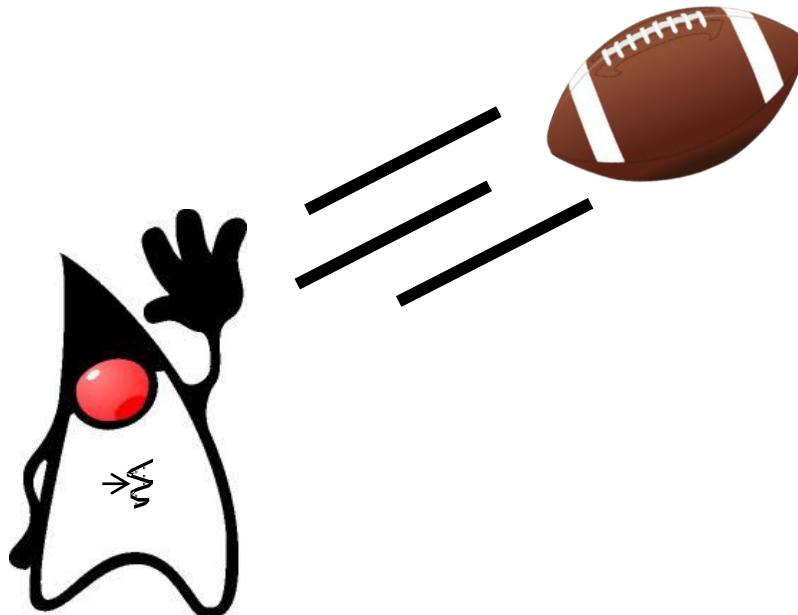
# Overview of the map() Intermediate Operation

- Applies a mapper function to every element of the input stream & returns an output stream consisting of the results



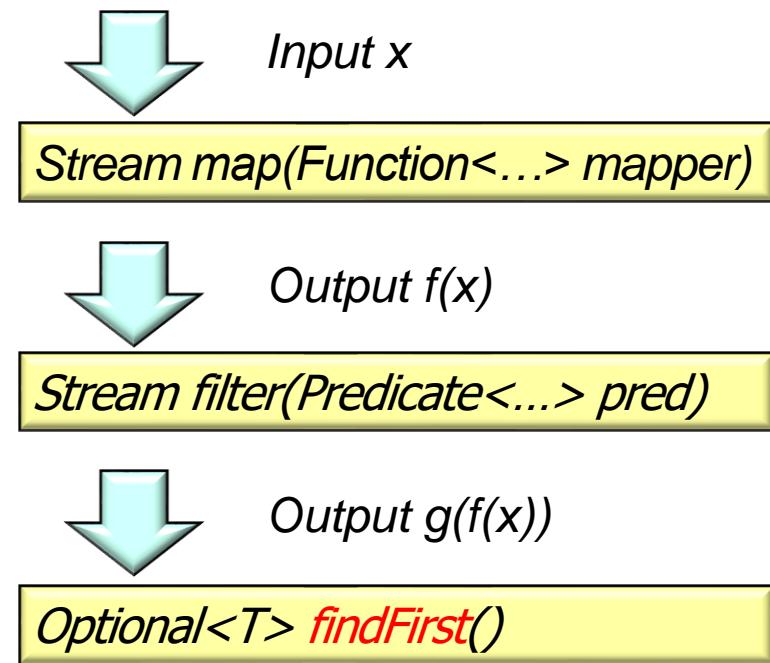
# Overview of the map() Intermediate Operation

- Applies a mapper function to every element of the input stream & returns an output stream consisting of the results
  - A mapper may throw an exception, which could terminate map()



# Overview of the map() Intermediate Operation

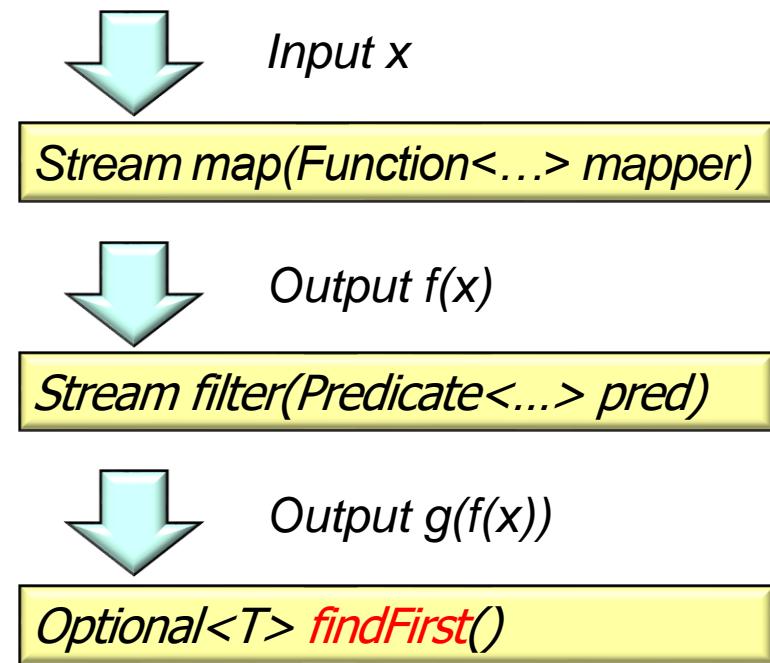
- Applies a mapper function to every element of the input stream & returns an output stream consisting of the results
  - A mapper may throw an exception, which could terminate map()
  - A short-circuit terminal operation also causes the map() operation to only process a subset of the input stream



# Overview of the map() Intermediate Operation

- Applies a mapper function to every element of the input stream & returns an output stream consisting of the results
  - A mapper may throw an exception, which could terminate map()
  - A short-circuit terminal operation also causes the map() operation to only process a subset of the input stream

ACROSS  
*the*  
BOARD



These caveats apply to all “run-to-completion” intermediate operations!

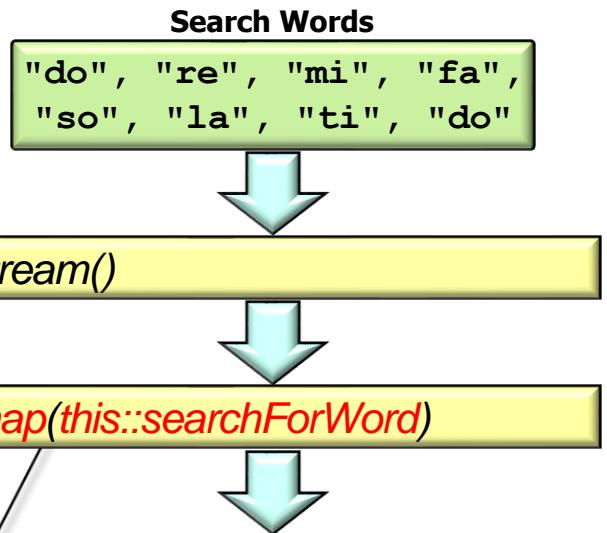
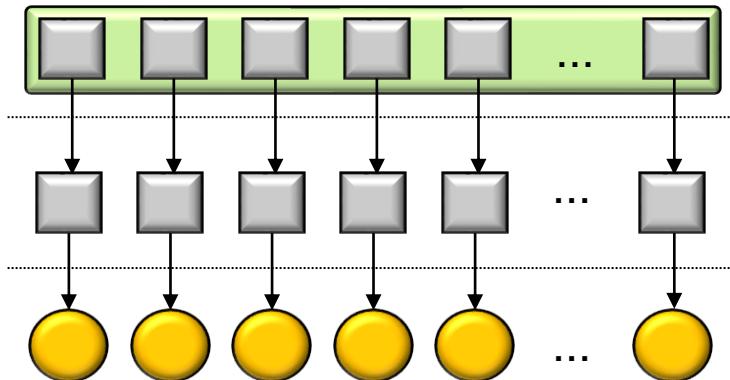
# Overview of the map() Intermediate Operation

- Example of applying map() & a mapper function in the SimpleSearchStream program

List  
<String>

Stream  
<String>

Stream  
<SearchResults>



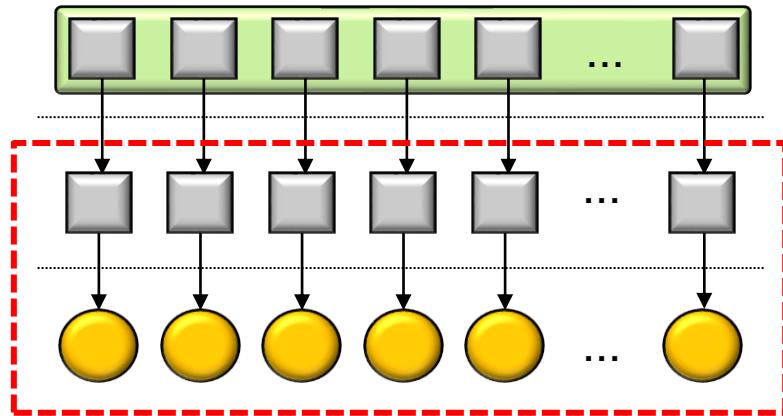
# Overview of the map() Intermediate Operation

- Example of applying map() & a mapper function in the SimpleSearchStream program

List  
<String>

Stream  
<String>

Stream  
<SearchResults>

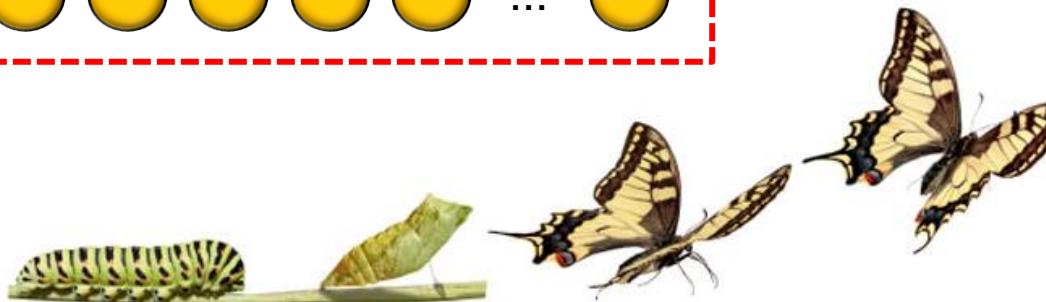


Search Words

```
"do", "re", "mi", "fa",
"so", "la", "ti", "do"
```

stream()

map(this::searchForWord)

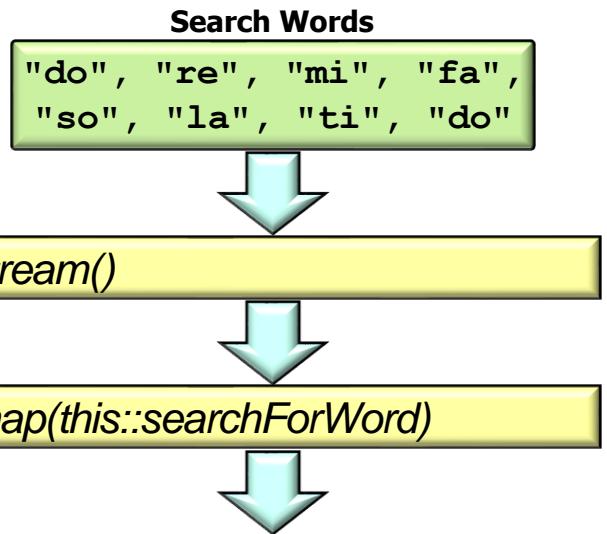


map() may transform the type of elements it processes

# Overview of the map() Intermediate Operation

- Example of applying map() & a mapper function in the SimpleSearchStream program

```
List<SearchResults> results =
 wordsToFind
 .stream()
 .map(this::searchForWord)
 .filter(not
 (SearchResults::isEmpty))
 .collect(toList());
```



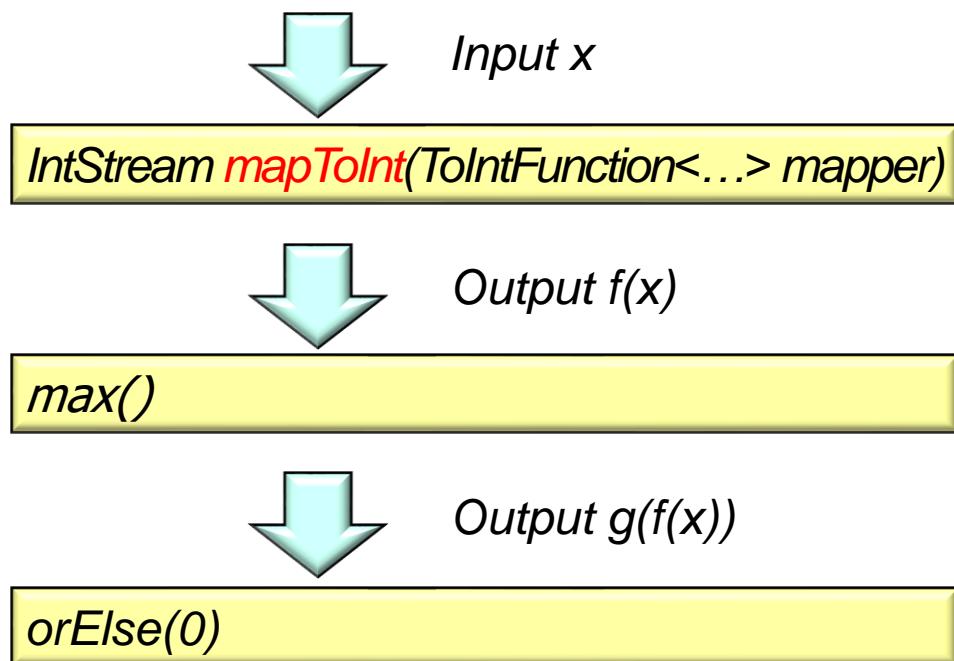
*Note "fluent" programming style with cascading method calls.*

---

# Overview of the mapToInt() Intermediate Operation

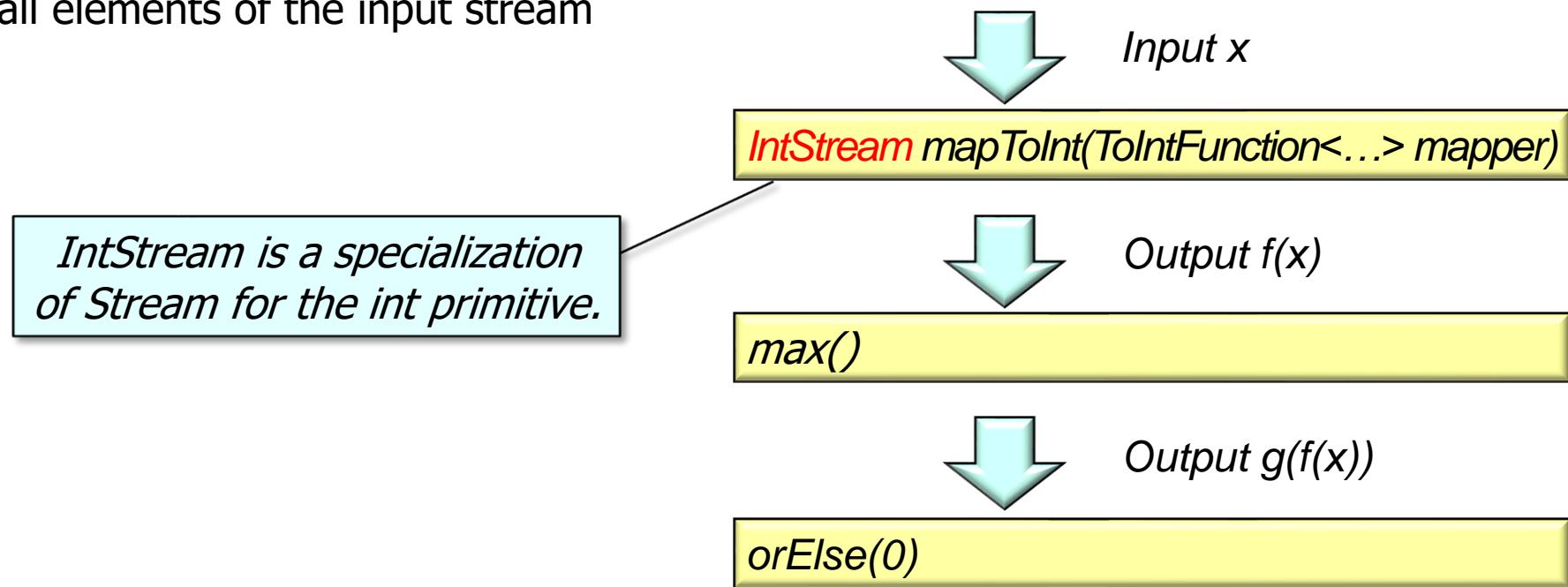
# Overview of the mapToInt() Intermediate Operation

- Returns an IntStream consisting of the results of applying the given mapper function to all elements of the input stream



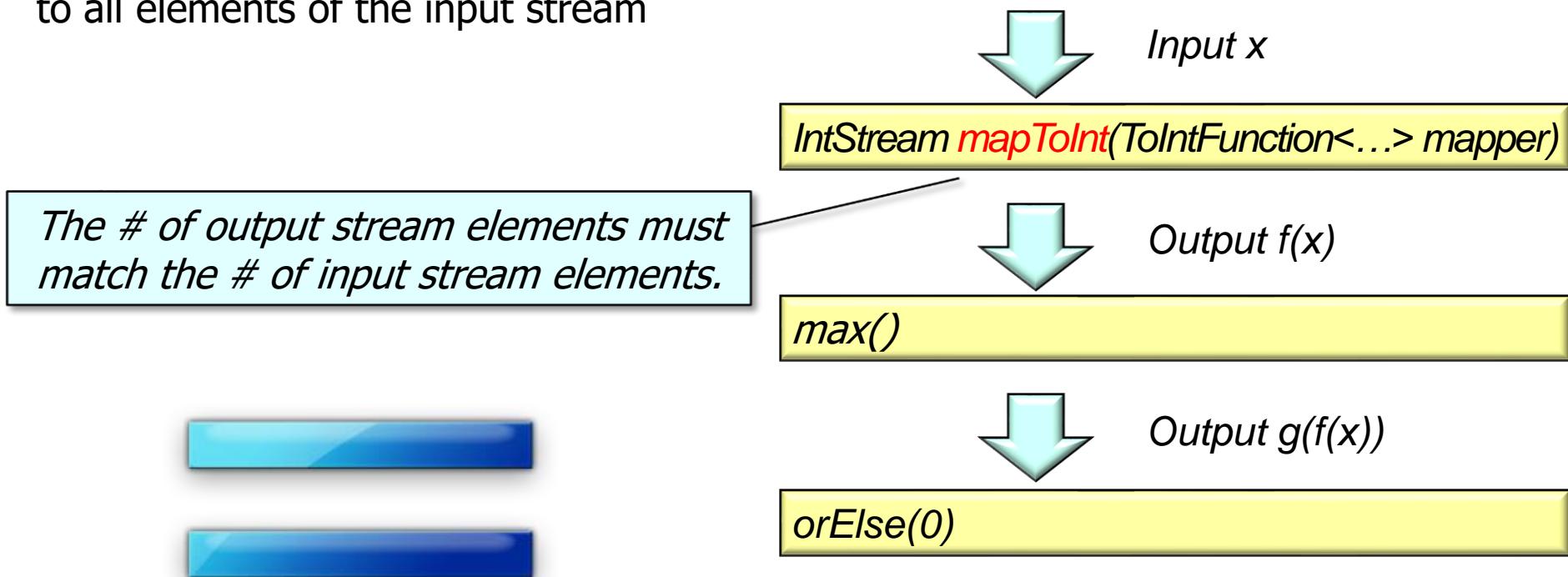
# Overview of the mapToInt() Intermediate Operation

- Returns an IntStream consisting of the results of applying the given mapper function to all elements of the input stream



# Overview of the mapToInt() Intermediate Operation

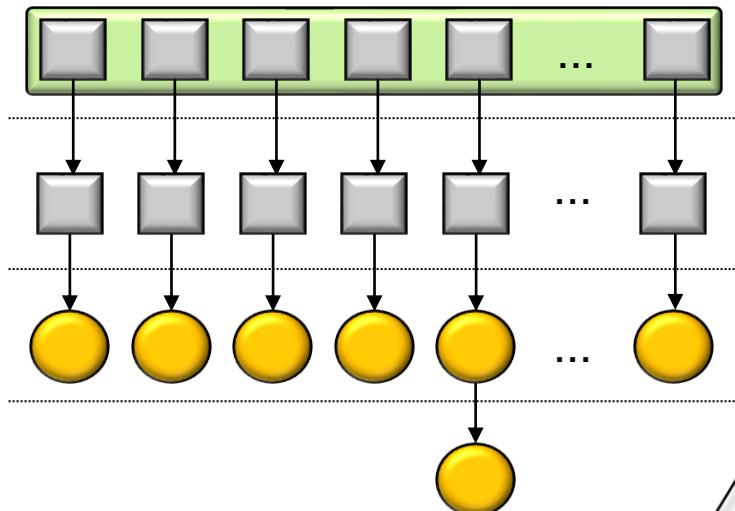
- Returns an IntStream consisting of the results of applying the given mapper function to all elements of the input stream



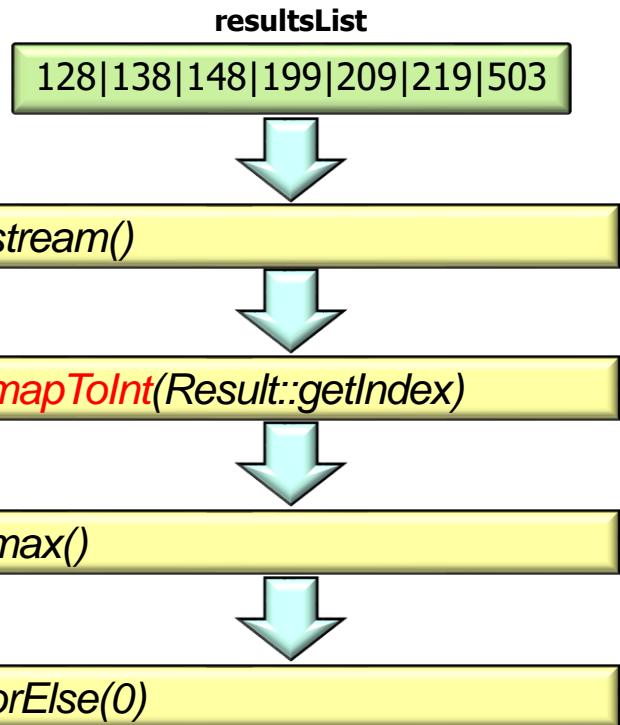
# Overview of the mapToInt() Intermediate Operation

- Example of applying mapToInt() & a mapper function in the SimpleSearchStream program

List  
<Result>  
  
Stream  
<Result>  
  
IntStream  
<int>  
  
OptionalInt



*Transform the stream of results into a stream of primitive int indices.*



# Overview of the mapToInt() Intermediate Operation

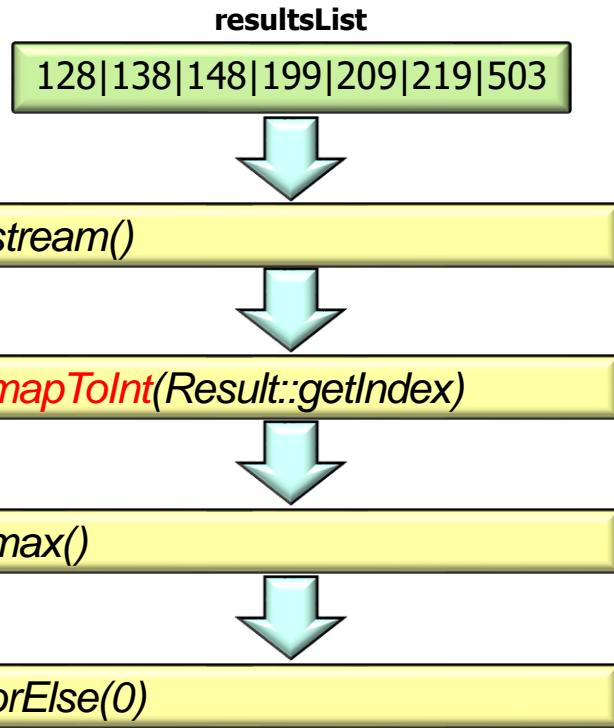
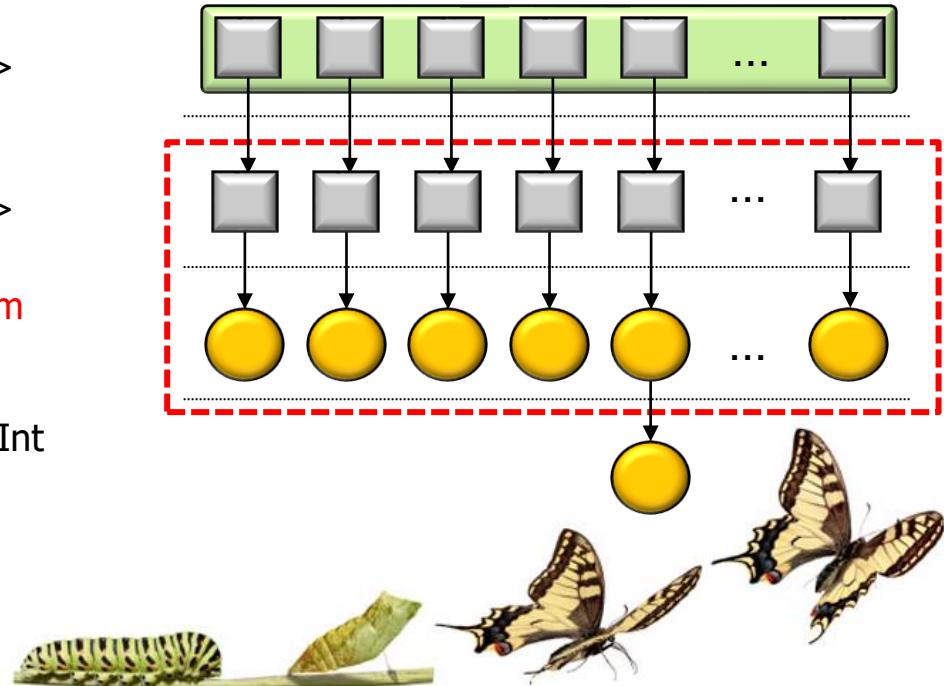
- Example of applying mapToInt() & a mapper function in the SimpleSearchStream program

List  
<Result>

Stream  
<Result>

IntStream  
<int>

OptionalInt



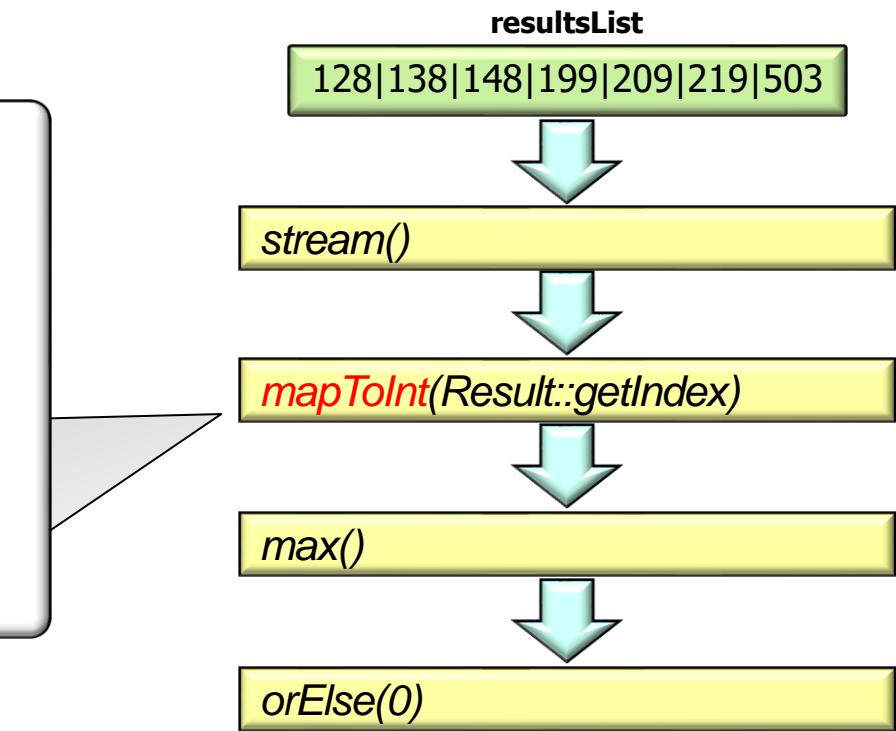
mapToInt() transforms the type of elements it processes into primitive ints

# Overview of the mapToInt() Intermediate Operation

- Example of applying mapToInt() & a mapper function in the SimpleSearchStream program

```
int computeMax
 (List<SearchResults.Result>
 resultsList) {
 return resultsList
 .stream()
 .mapToInt(SearchResults.Result
 ::getIndex)
 .max()
 .orElse(0);
}
```

*Note "fluent" programming style  
with cascading method calls.*



---

# End of Understand Java Streams

## Intermediate Operations

### map() & mapToInt()

# **Understand Java Streams Intermediate Operations filter() & flatMap()**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

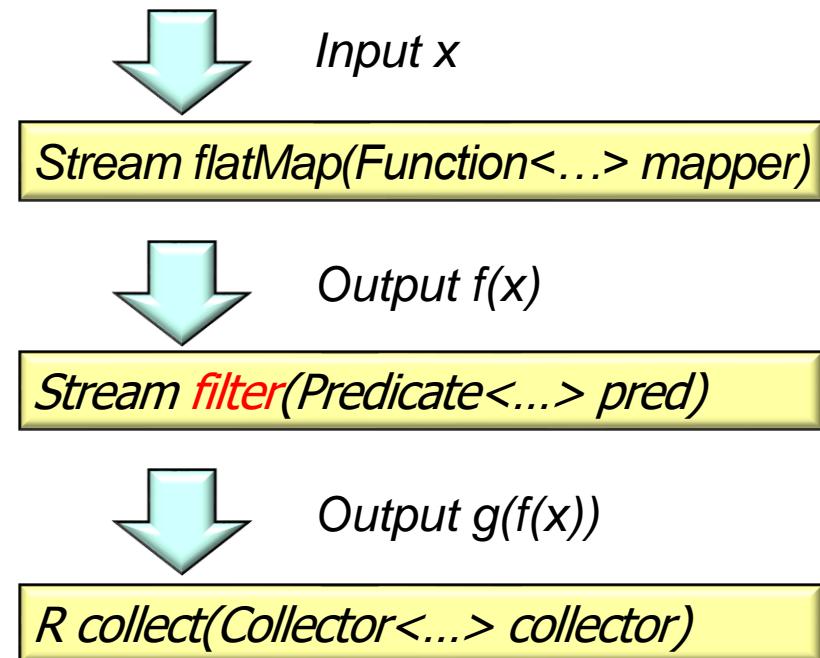
**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of stream aggregate operations
  - Intermediate operations
    - `map()` & `mapToInt()`
    - `filter()` & `flatMap()`



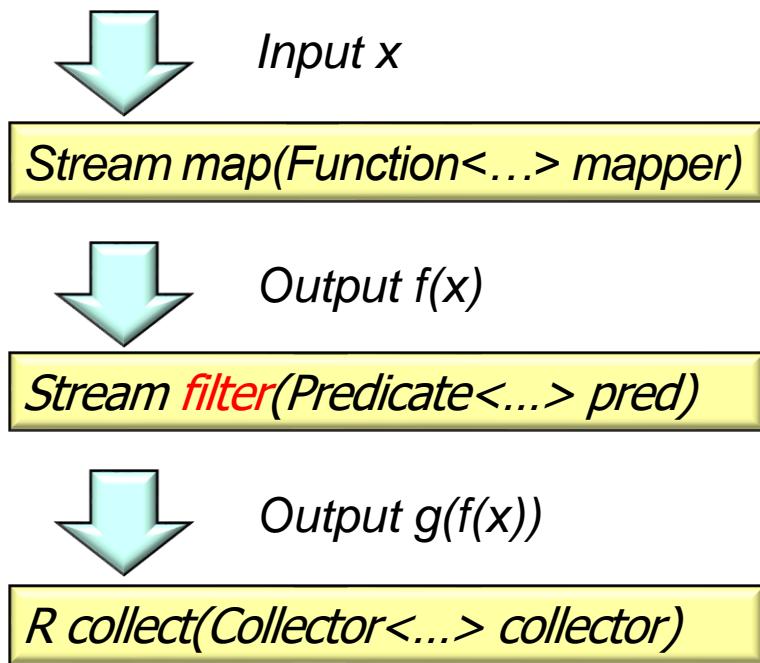
These are both stateless, run-to-completion operations

---

# Overview of the filter() Intermediate Operation

# Overview of the filter() Intermediate Operation

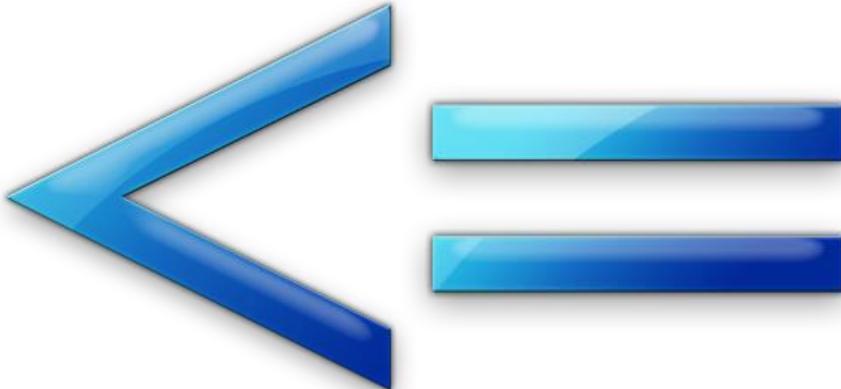
- Tests a predicate against each element of input stream & returns an output stream containing only elements that match the predicate



# Overview of the filter() Intermediate Operation

- Tests a predicate against each element of input stream & returns an output stream containing only elements that match the predicate

*The # of output stream elements may be less than the # of input stream elements.*



*Input x*



*Stream map(Function<...> mapper)*



*Output f(x)*

*Stream filter(Predicate<...> pred)*



*Output g(f(x))*

*R collect(Collector<...> collector)*

# Overview of the filter() Intermediate Operation

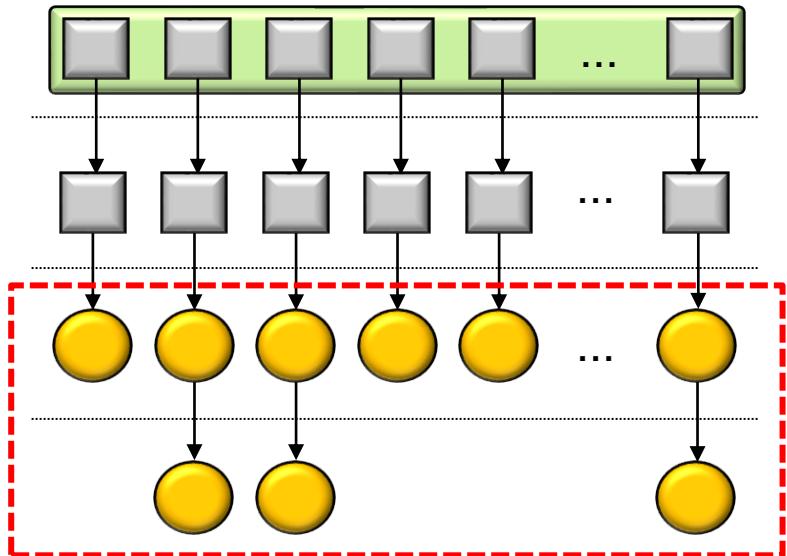
- Example of applying filter() & a predicate in the SimpleSearchStream program

List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>



Search Words

```
"do", "re", "mi", "fa",
"so", "la", "ti", "do"
```

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

*Filter out empty SearchResults.*

# Overview of the filter() Intermediate Operation

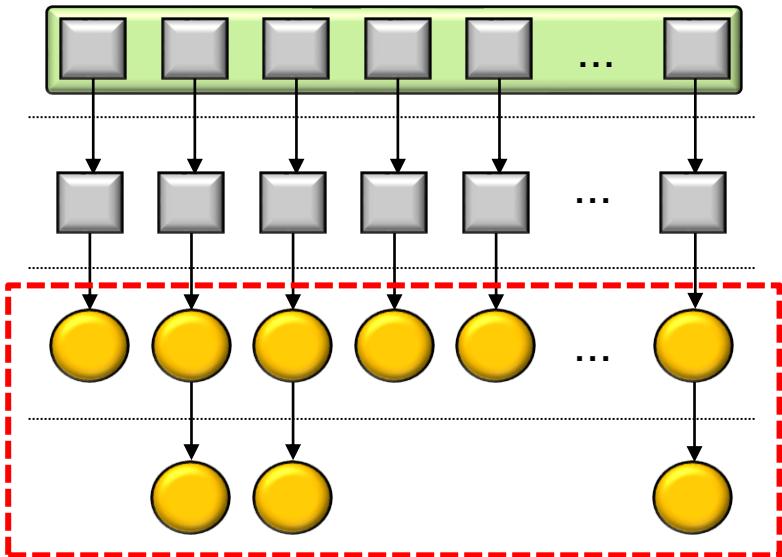
- Example of applying filter() & a predicate in the SimpleSearchStream program

List  
<String>

Stream  
<String>

Stream  
<SearchResults>

Stream  
<SearchResults>



Search Words

```
"do", "re", "mi", "fa",
"so", "la", "ti", "do"
```

stream()

map(this::searchForWord)

filter(not(SearchResults::isEmpty))

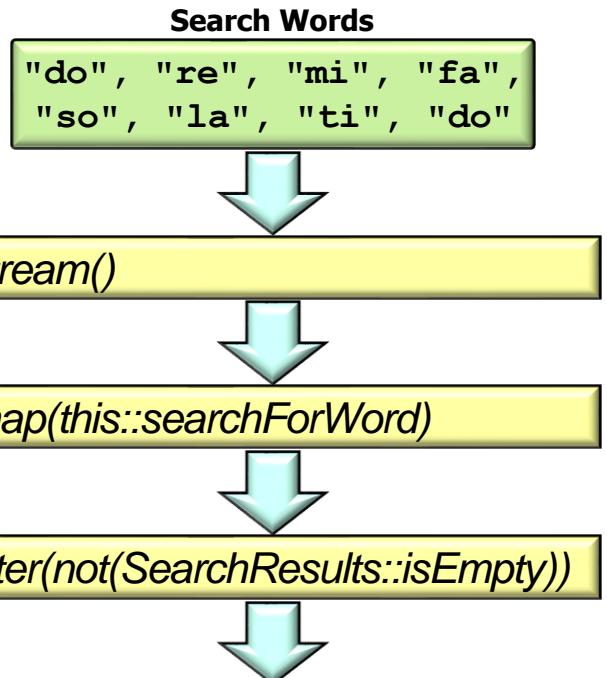


filter() can't change the type or value of elements it processes

# Overview of the filter() Intermediate Operation

- Example of applying filter() & a predicate in the SimpleSearchStream program

```
List<SearchResults> results =
 wordsToFind
 .stream()
 .map(this::searchForWord)
 .filter(not
 (SearchResults::isEmpty))
 .collect(toList());
```



*Again, note the fluent interface style.*

---

# Overview of the flatMap() Intermediate Operation

# Overview of the flatMap() Intermediate Operation

- Returns a stream that replaces each element of this stream w/contents of a mapped stream produced by applying the provided mapping function to each element

*This definition sounds like map() at first glance, but there are important differences!*

`Stream.of(I1, I2, I3, ..., In)`

*Output f(x)*

`flatMap(List::stream)`

*Output g(f(x))*

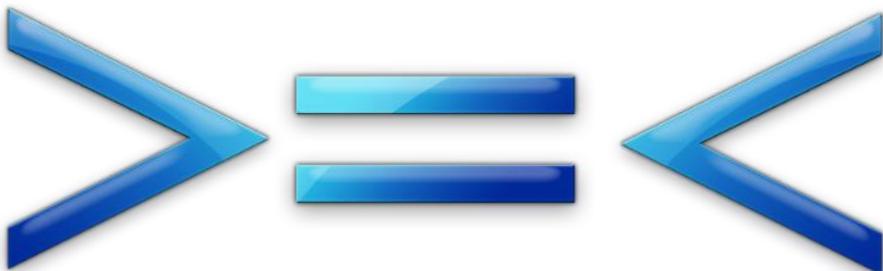
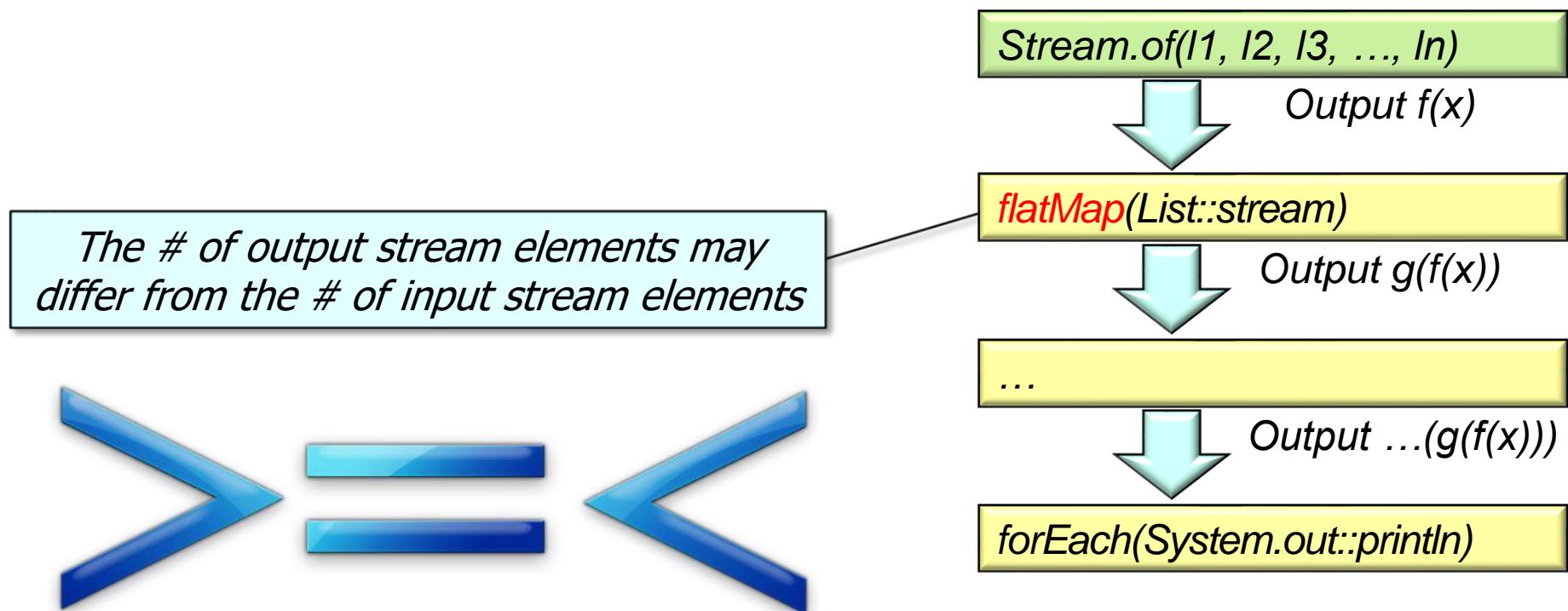
*...*

*Output ...(g(f(x)))*

`forEach(System.out::println)`

# Overview of the flatMap() Intermediate Operation

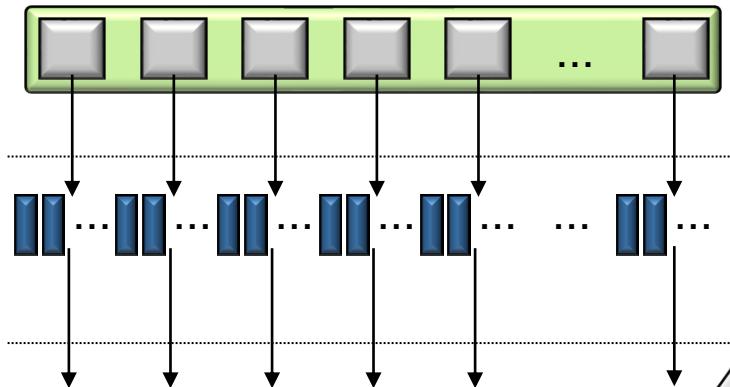
- Returns a stream that replaces each element of this stream w/contents of a mapped stream produced by applying the provided mapping function to each element



# Overview of the flatMap() Intermediate Operation

- Returns a stream that replaces each element of this stream w/contents of a mapped stream produced by applying the provided mapping function to each element

array<List  
<String>>



Stream.of(l1, l2, l3, ..., ln)

Output f(x)

flatMap(List::stream)

Output g(f(x))

...

Output ...(g(f(x)))

forEach(System.out::println)

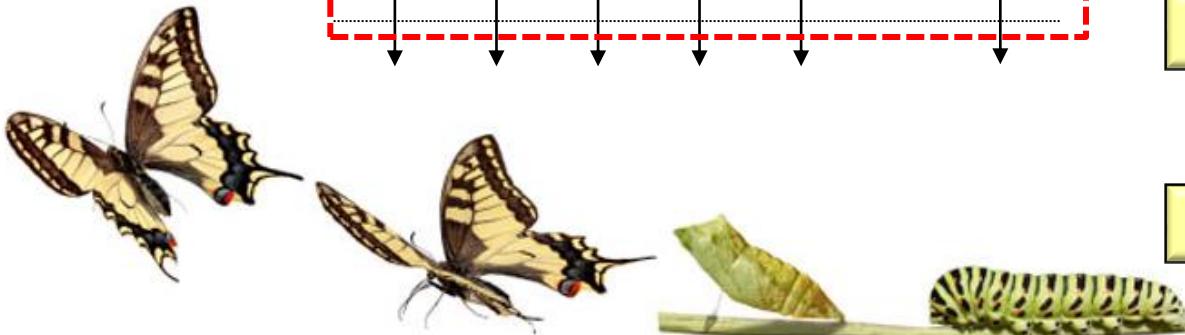
*"Flatten" an array of lists of strings into a stream of strings*

# Overview of the flatMap() Intermediate Operation

- Returns a stream that replaces each element of this stream w/contents of a mapped stream produced by applying the provided mapping function to each element

array<List  
<String>>

Stream  
<String>



Stream.of(I1, I2, I3, ..., In)

Output  $f(x)$

flatMap(List::stream)

Output  $g(f(x))$

...

Output ...( $g(f(x))$ )

forEach(System.out::println)

flatMap() *may* transform the type of elements it processes

# Overview of the flatMap() Intermediate Operation

- Returns a stream that replaces each element of this stream w/contents of a mapped stream produced by applying the provided mapping function to each element

```
List<String> l1 = ...;
List<String> l2 = ...;
List<String> l3 = ...;
...
List<String> ln = ...;
```

```
Stream
.of(l1, l2, l3, ..., ln)
.flatMap(List::stream)
...
.forEach(System.out::println);
```

Stream.of(l1, l2, l3, ..., ln)

Output  $f(x)$

flatMap(List::stream)

Output  $g(f(x))$

...

Output  $\dots(g(f(x)))$

forEach(System.out::println)

---

# End of Understand Java Streams

## Intermediate Operations

### filter() & flatMap()

# **Understand Java Streams**

## **Terminal Operations**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

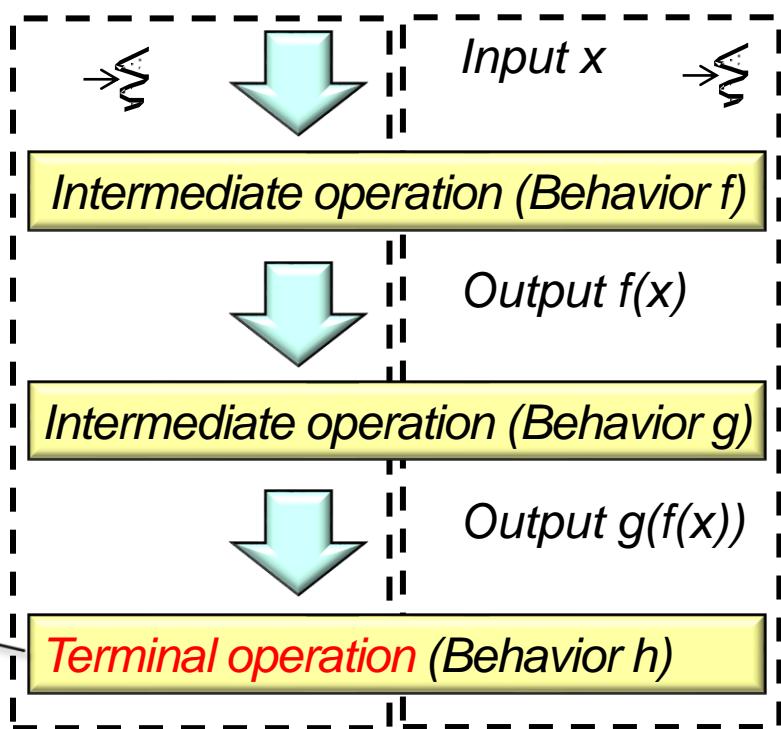
---

- Understand the structure & functionality of stream terminal operations



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of stream terminal operations



*These operations also apply to both sequential & parallel streams*

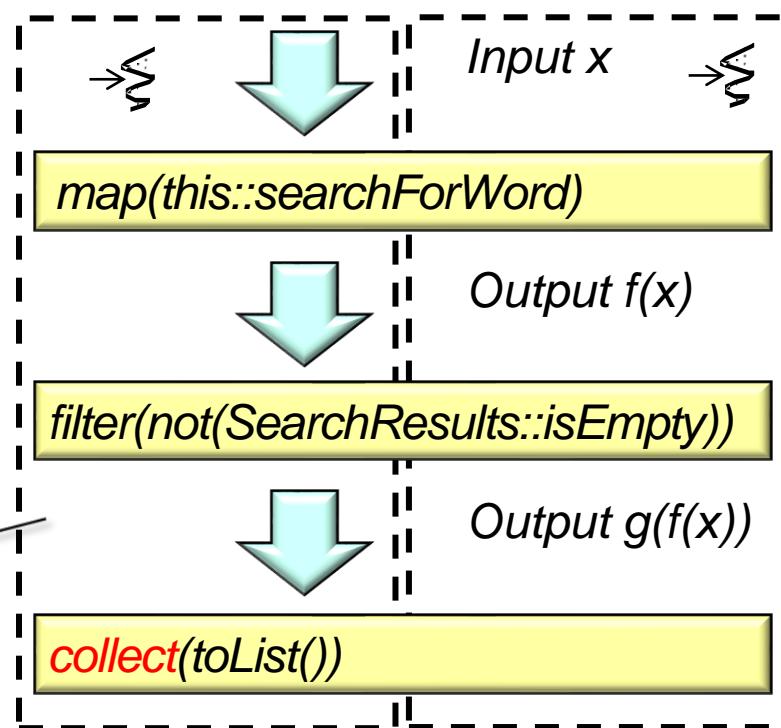
# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of stream terminal operations

**Input String to Search**  
Let's start at the very beginning..

**Search Words**  
"do", "re", "mi", "fa",  
"so", "la", "ti", "do"

We continue to showcase the SimpleSearchStream program



---

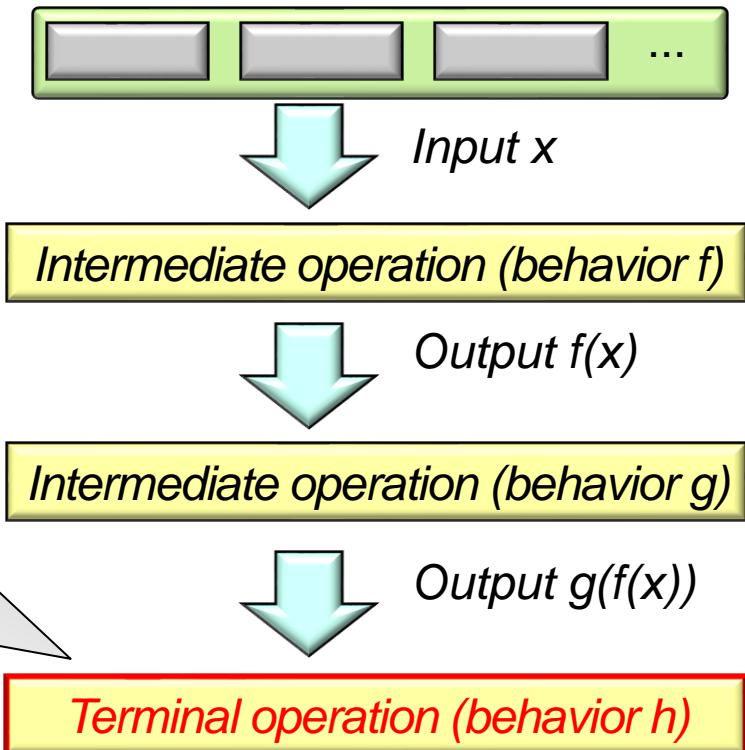
# Overview of Terminal Operations

# Overview of Common Stream Terminal Operations

- Every stream finishes with a terminal operation that yields a non-stream result

Stream

```
.of("horatio",
 "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```



# Overview of Common Stream Terminal Operations

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - No value at all
    - e.g., `forEach()` & `forEachOrdered()`

*`forEach()` & `forEachOrdered()`  
only have side-effects!*



# Overview of Common Stream Terminal Operations

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.

- No value at all
  - e.g., `forEach()` & `forEachOrdered()`

```
Stream
 .of("horatio",
 "laertes",
 "Hamlet", ...)
 .filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
 .map(this::capitalize)
 .sorted()
 .forEach
 (System.out::println);
```

*Print each character in Hamlet that starts with 'H' or 'h' in consistently capitalized & sorted order.*

# Overview of Common Stream Terminal Operations

---

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - No value at all
  - The result of a reduction operation
    - e.g., `collect()` & `reduce()`



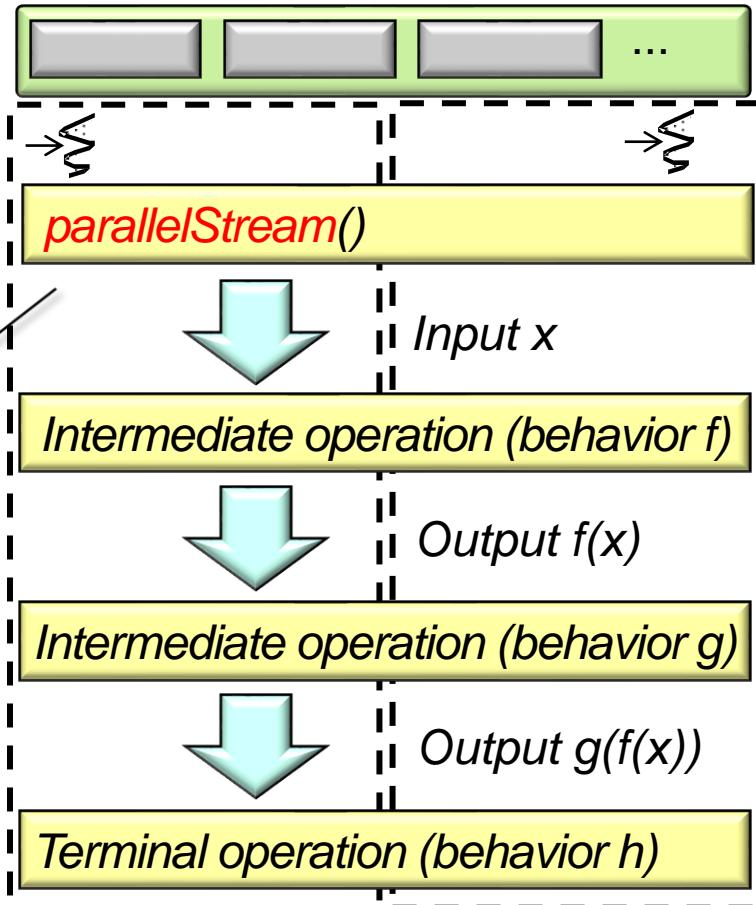
---

See [docs.oracle.com/javase/tutorial/collectionsstreams/reduction.html](https://docs.oracle.com/javase/tutorial/collectionsstreams/reduction.html)

# Overview of Common Stream Terminal Operations

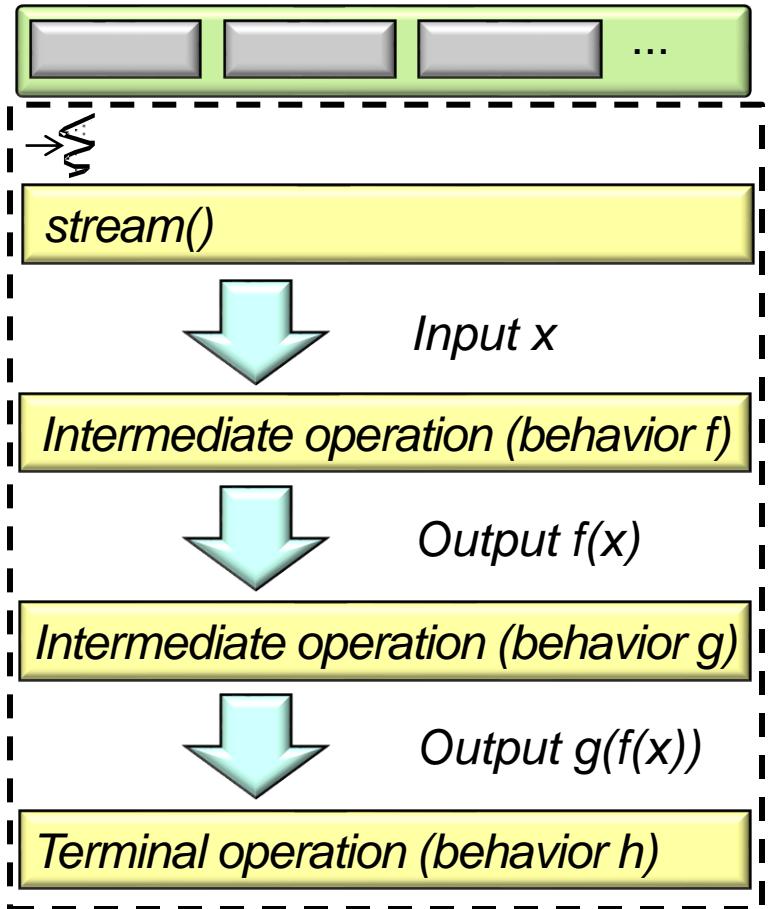
- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - No value at all
  - The result of a reduction operation
    - e.g., `collect()` & `reduce()`

*`collect()` & `reduce()` terminal operations work seamlessly with parallel streams.*



# Overview of the collect() Terminal Operation

- A terminal operation also triggers all the ("lazy") intermediate operation processing



---

# End of Understand Java Streams Terminal Operations

# **Understand the Java Streams forEach() Terminal Operation**

**Douglas C. Schmidt**

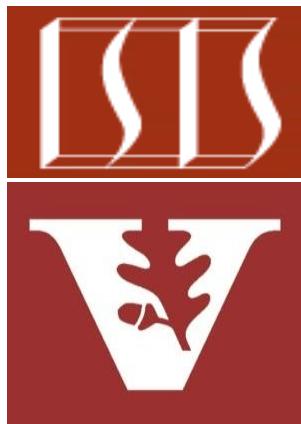
**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand common terminal operations, e.g.

- `forEach()`

```
void runForEach() {
 ...
}
```

*We showcase `forEach()`  
using the Hamlet program*

`Stream`

```
.of("horatio", "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach
 (System.out::println);
...
```

---

# Stream Terminal Operations That Return No Value

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all

```
void runForEach() {
 ...

 Stream
 .of("horatio", "laertes",
 "Hamlet", ...)
 .filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
 .map(this::capitalize)
 .sorted()
 .forEach
 (System.out::println);
 ...
}
```

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all

*Several variants of forEach()  
are showcased in this example.*

```
void runForEach() {
 ...
}
```

Stream

```
.of("horatio", "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach
 (System.out::println);
...
```

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all

```
void runForEach() {
 ...
}
```

Create & process a stream consisting  
of characters from the play "Hamlet"

Stream

```
.of("horatio", "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach
 (System.out::println);
...
```

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all

```
void runForEach() {
 ...
}
```

Stream

```
.of("horatio", "laertes",
 "Hamlet", ...)
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach
(System.out::println);
...
```

*Performs the designated action  
on each element of this stream*

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all

```
void runForEach() {
 List<String> results =
 new ArrayList<>();

 Stream
 .of("horatio", "laertes",
 "Hamlet", ...)
 .filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
 .map(this::capitalize)
 .sorted()
 .forEach
 (results::add);
 ...
}
```

*The method reference passed to forEach() can have side-effects*

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all

*Using forEach() in this manner is tricky, however, since programmers must remember to initialize the results object!*



```
void runForEach() {
 List<String> results =
 new ArrayList<>();

 Stream
 .of("horatio", "laertes",
 "Hamlet", ...)
 .filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
 .map(this::capitalize)
 .sorted()
 .forEach
 (results::add);

 ...
}
```

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all

*Likewise, using forEach() with side-effects in a parallel stream can incur nasty race conditions!!*



```
void runForEach() {
 List<String> results =
 new ArrayList<>();

 Stream
 .of("horatio", "laertes",
 "Hamlet", ...)
 .parallel()
 .filter(s -> s.toLowerCase()
 .charAt(0) == 'h')
 .map(s -> s.toUpperCase())
 .sorted()
 .forEach
 (results::add);
 ...
}
```

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all

```
void runForEach() {
 Queue<String> results = new
 ConcurrentLinkedQueue<>();

 Stream
 .of("horatio", "laertes",
 "Hamlet", ...)
 .parallel()
 .filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
 .map(this::capitalize)
 .sorted()
 .forEach
 (results::add);
 ...
}
```

*ConcurrentLinkedQueue could be used here, but it's still error-prone & inefficient*

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all



*Use `forEachOrdered()` if presenting the results in "encounter order" is important.*

```
void runForEach() {
 ...

 Stream
 .of("horatio", "laertes",
 "Hamlet", ...)
 .parallel()
 .filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
 .map(this::capitalize)
 .sorted()
 .forEachOrdered
 (System.out::println);
 ...
}
```

# Stream Terminal Operations That Return No Value

- Several terminal operations return no value at all

```
void runForEach() {
 ...
}
```

Stream

```
.of("horatio", "laertes",
 "Hamlet", ...)
.parallel()
.filter(s -> toLowerCase
 (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEachOrdered
 (System.out::println);
...
```

*forEachOrdered() is only really relevant for parallel streams..*

---

End of Understand the  
Java Streams forEach()  
Terminal Operation

# **Understand the Java Streams collect() Terminal Operation**

**Douglas C. Schmidt**

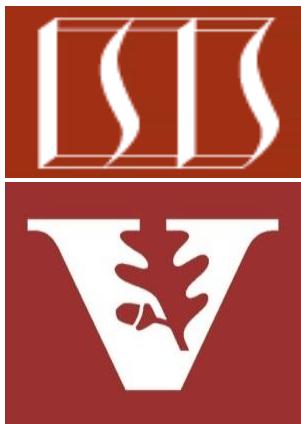
**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

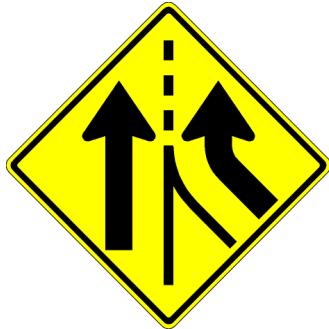
**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand common terminal operations, e.g.

- `forEach()`
- `collect()`



*We showcase `collect()`  
using the Hamlet program*

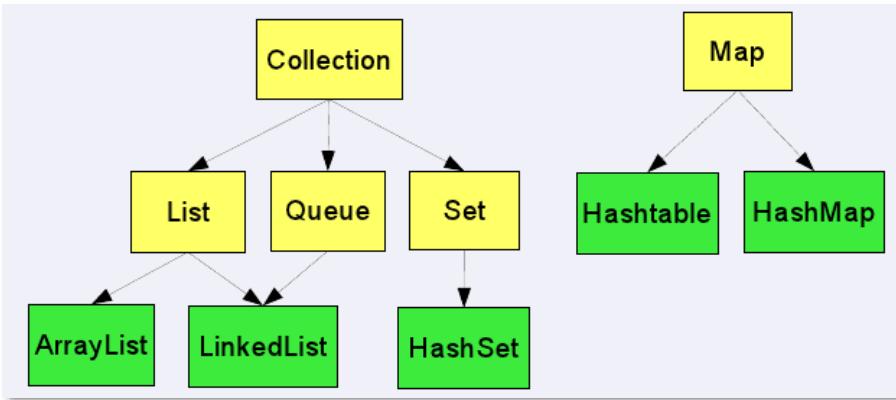
```
void runCollectTo* () {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);
 ...<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...) == 'h')
 .map(this::capitalize)
 .sorted()
 .collect(...); ...
```

---

# A Stream Terminal Operation That Returns Collections

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



```
void runCollectTo* () {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);
 ...<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...).=='h')
 .map(this::capitalize)
 .sorted()
 .collect(...); ...
}
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection

*Many variants of collect() are showcased in this example.*

```
void runCollectTo*() {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);
 ...<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...) == 'h')
 .map(this::capitalize)
 .sorted()
 .collect(...); ...
}
```

# A Stream Terminal Operation That Returns Collections

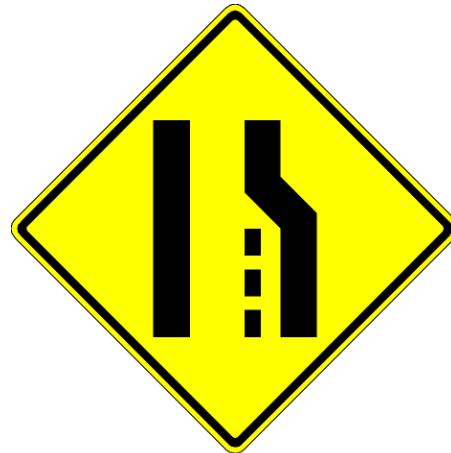
- The collect() terminal operation typically returns a collection

*Create & process a stream consisting of characters from the play "Hamlet".*

```
void runCollectTo* () {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);
 ...<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...) == 'h')
 .map(this::capitalize)
 .sorted()
 .collect(...); ...
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



*Performs a mutable reduction on all elements of this stream using some collector & returns a single result.*

```
void runCollectTo* () {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);
 ...<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...). == 'h')
 .map(this::capitalize)
 .sorted()
 .collect(...); ...
}
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



*A collector performs reduction operations, e.g., summarizing elements according to various criteria, accumulating elements into various types of collections, etc.*

```
void runCollectTo* () {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);
 ...<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...). == 'h')
 .map(this::capitalize)
 .sorted()
 .collect(...); ...
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



```
void runCollectToList() {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet, . . .");

 List<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...). == 'h')
 .map(this::capitalize)
 .sorted()
 .collect(toList()); . . .
```

*Collect results into a ArrayList,  
which can contain duplicates.*

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



```
void runCollectToList() {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet, . . .");

 List<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...).=='h')
 .map(this::capitalize)
 .sorted()
 .collect(toList()); . . .
```

*collect() is much less error-prone than forEach()  
since initialization is implicit & it's thread-safe.*

See earlier lesson on “Java Streams: the forEach() Terminal Operation”

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



*Collect the results into a HashSet, which can contain no duplicates.*

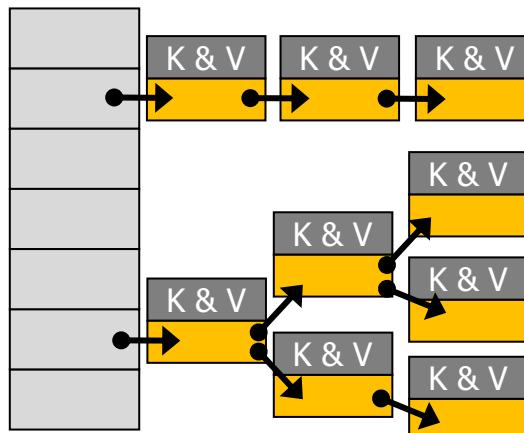
```
void runCollectToSet() {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);

 Set<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...).=='h')
 .map(this::capitalize)
 .collect(toSet()); ...
}
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection

*Array    Linked lists/trees*



*Collect results into a HashMap, along with the length of (merged duplicate) entries.*

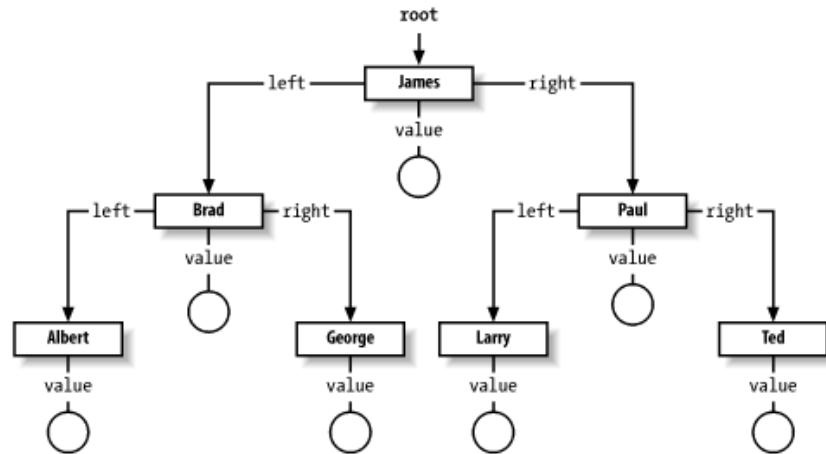
```
void runCollectToMap() {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);

 Map<String, Integer> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(...).equals('h'))
 .map(this::capitalize)
 .collect(toMap(identity(),
 String::length,
 Integer::sum));
}
```

...

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



*Collect the results into a TreeMap by grouping elements according to name (key) & name length (value).*

```
void runCollectGroupingBy() {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);

 Map<String, Long> results =
 ...
 .collect
 (groupingBy
 (identity(),
 TreeMap::new,
 summingLong
 (String::length)));
 ...
}
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



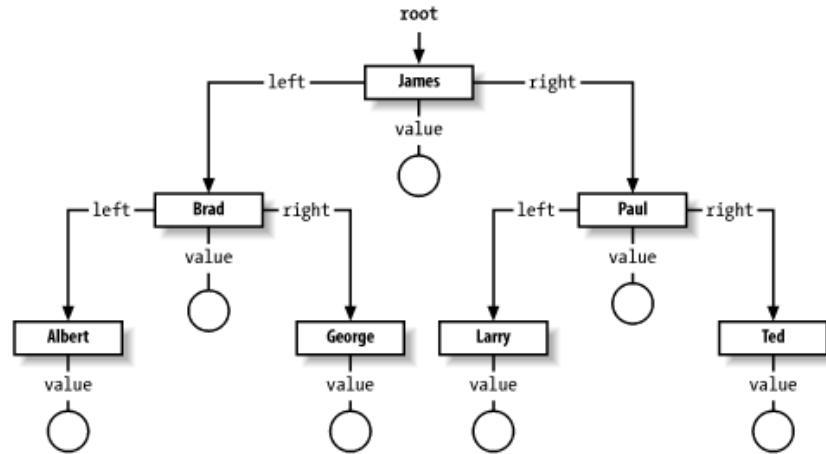
*groupingBy() partitions a stream via a "classifier" function (identity() always returns its input argument).*

```
void runCollectGroupingBy() {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);

 Map<String, Long> results =
 ...
 .collect
 (groupingBy
 (identity(),
 TreeMap::new,
 summingLong
 (String::length)));
 ...
}
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



*A constructor reference is used to create a TreeMap.*

```
void runCollectGroupingBy() {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);

 Map<String, Long> results =
 ...
 .collect
 (groupingBy
 (identity(),
 TreeMap::new,
 summingLong
 (String::length)));
 ...
}
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection

*This "downstream collector" defines a summingLong() collector that's applied to the results of the classifier function.*

```
void runCollectGroupingBy() {
 List<String> characters =
 List.of("horatio",
 "laertes",
 "Hamlet", ...);

 Map<String, Long> results =
 ...
 .collect
 (groupingBy
 (identity(),
 TreeMap::new,
 summingLong
 (String::length)));
 ...
}
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection

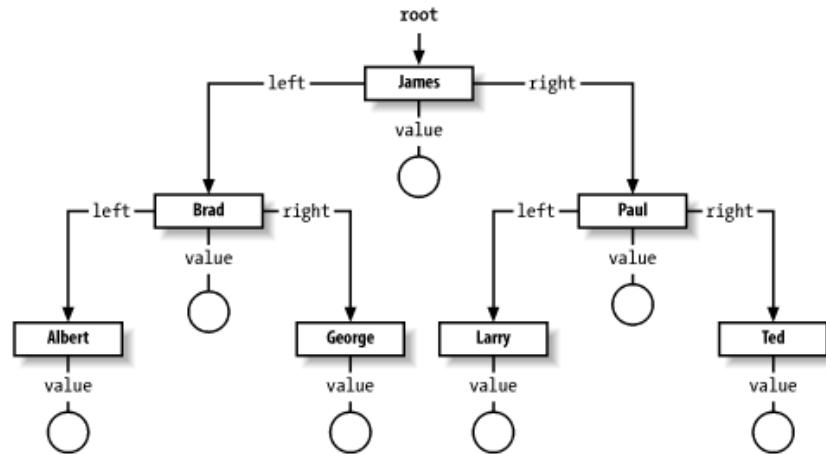


*Convert a string into a stream via regular expression splitting!*

```
void runCollectReduce() {
 Map<String, Long>
 matchingCharactersMap =
 Pattern.compile(",")
 .splitAsStream
 ("horatio,Hamlet,...")
 ...
 .collect
 (groupingBy
 (identity(),
 TreeMap::new,
 summingLong
 (String::length))) ;
```

# A Stream Terminal Operation That Returns Collections

- The collect() terminal operation typically returns a collection



*Collect the results into a TreeMap by grouping elements according to name (key) & name length (value).*

```
void runCollectReduce() {
 Map<String, Long>
 matchingCharactersMap =
 Pattern.compile(",")
 .splitAsStream
 ("horatio,Hamlet,...")
 ...
 .collect
 (groupingBy
 (identity(),
 TreeMap::new,
 summingLong
 (String::length)));
}
```

---

# End of Understand the Java Streams collect() Terminal Operation

# **Understand the Java Streams reduce() Terminal Operation**

**Douglas C. Schmidt**

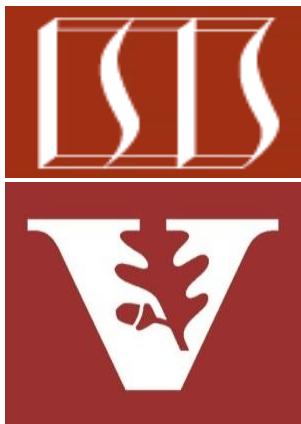
**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand common terminal operations, e.g.

- `forEach()`
- `collect()`
- `reduce()`



```
void runCollectReduce () {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .reduce(0L,
 Long::sum);
```

*We showcase `reduce()`  
using the Hamlet program*

---

# A Stream Terminal Operation That Returns a Primitive

# A Stream Terminal Operation That Returns a Primitive

- The reduce() terminal operation typically returns a primitive value



```
void runCollectReduce1() {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .reduce(0L,
 Long::sum);
```

# A Stream Terminal Operation That Returns a Primitive

- The reduce() terminal operation typically returns a primitive value

*Create a map associating the names of Hamlet characters with their name lengths.*

```
void runCollectReduce1() {
 Map<String, Long>
 matchingCharactersMap =
 ...
 .collect
 (groupingBy
 (identity(),
 TreeMap::new,
 summingLong
 (String::length)));
```

# A Stream Terminal Operation That Returns a Primitive

- The reduce() terminal operation typically returns a primitive value

```
void runCollectReduce1() {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .reduce(0L,
 Long::sum);
```

*Convert the map's values list into a stream of long values.*

# A Stream Terminal Operation That Returns a Primitive

- The reduce() terminal operation typically returns a primitive value

```
void runCollectReduce1() {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .reduce(0L,
 Long::sum);
```

*Sum up the lengths of all character names in Hamlet.*

# A Stream Terminal Operation That Returns a Primitive

- The reduce() terminal operation typically returns a primitive value

```
void runCollectReduce1() {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .reduce(0L,
 Long::sum);
```

*0 is the "identity," i.e., the initial value of the reduction & the default result if there are no elements in the stream.*

# A Stream Terminal Operation That Returns a Primitive

- The reduce() terminal operation typically returns a primitive value



```
void runCollectReduce1() {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .reduce(0L,
 Long::sum);
```

*This method reference is an "accumulator," which is a stateless function that combines two values into a single (immutable) "reduced" value.*

See [docs.oracle.com/javase/8/docs/api/java/lang/Long.html#sum](https://docs.oracle.com/javase/8/docs/api/java/lang/Long.html#sum)

# A Stream Terminal Operation That Returns a Primitive

- The reduce() terminal operation typically returns a primitive value

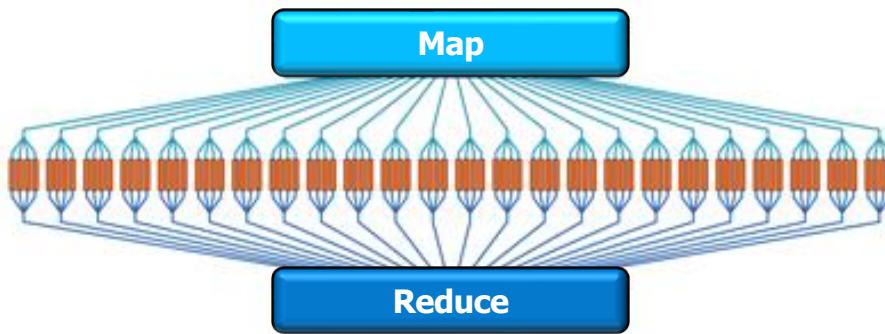
```
void runCollectReduce1() {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .reduce(0L,
 (x, y) -> x + y);
```

*A lambda expression could also be used here.*

# A Stream Terminal Operation That Returns a Primitive

- The three parameter “map/reduce” version of reduce() is used along with parallel streams



```
void runCollectMapReduce() {
 List<String> characterList =
 ...

 long sumOfNameLengths =
 characterList
 .parallelStream()
 .reduce(0L,
 (sum, s) ->
 sum + s.length(),
 Long::sum);
}
```

# A Stream Terminal Operation That Returns a Primitive

- The three parameter “map/reduce” version of reduce() is used along with parallel streams

*Generate a consistently capitalized & sorted list of names of Hamlet characters starting with the letter 'h'.*

```
void runCollectMapReduce() {
 List<String> characterList =
 ...

 long sumOfNameLengths =
 characterList
 .parallelStream()
 .reduce(0L,
 (sum, s) ->
 sum + s.length(),
 Long::sum);
}
```

# A Stream Terminal Operation That Returns a Primitive

- The three parameter “map/reduce” version of reduce() is used along with parallel streams

```
void runCollectMapReduce() {
 List<String> characterList =
 ...

 long sumOfNameLengths =
 characterList
 .parallelStream()
 .reduce(0L,
 (sum, s) ->
 sum + s.length(),
 Long::sum);
}
```

*Convert the list into a parallel stream.*

# A Stream Terminal Operation That Returns a Primitive

- The three parameter “map/reduce” version of reduce() is used along with parallel streams

```
void runCollectMapReduce() {
 List<String> characterList =
 ...

 long sumOfNameLengths =
 characterList
 .parallelStream()
 .reduce(0L,
 (sum, s) ->
 sum + s.length(),
 Long::sum);
}
```

*Perform a reduction on the stream with an initial value of 0.*

# A Stream Terminal Operation That Returns a Primitive

- The three parameter “map/reduce” version of reduce() is used along with parallel streams

```
void runCollectMapReduce() {
 List<String> characterList =
 ...

 long sumOfNameLengths =
 characterList
 .parallelStream()
 .reduce(0L,
 (sum, s) ->
 sum + s.length(),
 Long::sum);
}
```

*This lambda expression is an accumulator that performs the “map” operation.*

# A Stream Terminal Operation That Returns a Primitive

- The three parameter “map/reduce” version of reduce() is used along with parallel streams

```
void runCollectMapReduce() {
 List<String> characterList =
 ...

 long sumOfNameLengths =
 characterList
 .parallelStream()
 .reduce(0L,
 (sum, s) ->
 sum + s.length(),
 Long::sum);
```

*This method reference performs the "reduce" operation.*

# A Stream Terminal Operation That Returns a Primitive

- The sum() terminal operation avoids the need to use reduce()

```
void runCollectReduce2 () {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .mapToLong (Long::longValue)
 .sum();
```

# A Stream Terminal Operation That Returns a Primitive

- The sum() terminal operation avoids the need to use reduce()

```
void runCollectReduce2 () {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .mapToLong (Long::longValue)
 .sum();
```

*Convert the map into a stream of long values.*

# A Stream Terminal Operation That Returns a Primitive

- The sum() terminal operation avoids the need to use reduce()

```
void runCollectReduce2 () {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .mapToLong (Long::longValue)
 .sum();
```

*Map the stream of Long objects into a stream of long primitives.*

# A Stream Terminal Operation That Returns a Primitive

- The sum() terminal operation avoids the need to use reduce()

```
void runCollectReduce2 () {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .mapToLong (Long::longValue)
 .sum();
```

*Sum the stream of long primitives into a single result.*

# A Stream Terminal Operation That Returns a Primitive

- `collect()` can also be used to return a primitive value

```
void runCollectReduce3() {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .collect
 (summingLong
 (Long::longValue));
```

See earlier lesson on “Java Streams: the `collect()` Terminal Operation”

# A Stream Terminal Operation That Returns a Primitive

- `collect()` can also be used to return a primitive value

```
void runCollectReduce3() {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .collect
 (summingLong
 (Long::longValue));
```

*Trigger the stream processing.*

# A Stream Terminal Operation That Returns a Primitive

- `collect()` can also be used to return a primitive value

```
void runCollectReduce3() {
 Map<String, Long>
 matchingCharactersMap =
 ...

 long sumOfNameLengths =
 matchingCharactersMap
 .values()
 .stream()
 .collect
 (summingLong
 (Long::longValue));
```

*Return a collector that produces the sum of a long-value function applied to input elements.*

# A Stream Terminal Operation That Returns a Primitive

---

- `reduce()` can also be used to return a non-primitive value

```
void streamReduceConcat
 (boolean parallel) {
 ...
 Stream<String> wordStream =
 allWords.stream();

 ...

 String words = wordStream
 .reduce(new String() ,
 (x, y) -> x + y);
```

# A Stream Terminal Operation That Returns a Primitive

- `reduce()` can also be used to return a non-primitive value

```
void streamReduceConcat
 (boolean parallel) {
 ...
 Stream<String> wordStream =
 allWords.stream();

 ...

 String words = wordStream
 .reduce(new String(),
 (x, y) -> x + y);
```

*reduce() creates a string containing all the concatenated words in a stream*

# A Stream Terminal Operation That Returns a Primitive

- `reduce()` can also be used to return a non-primitive value

```
void streamReduceConcat
 (boolean parallel) {
 ...
 Stream<String> wordStream =
 allWords.stream();

 ...

 String words = wordStream
 .reduce(new String() ,
 (x, y) -> x + y);
```

*This simple fix is inefficient due to string concatenation overhead*

---

# End of Understand the Java Streams reduce() Terminal Operation

# Contrast the Java Streams reduce() & collect() Terminal Operations

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Understand common terminal operations, e.g.
  - `forEach()`
  - `collect()`
  - `reduce()`
  - Contrasting `reduce()` & `collect()`

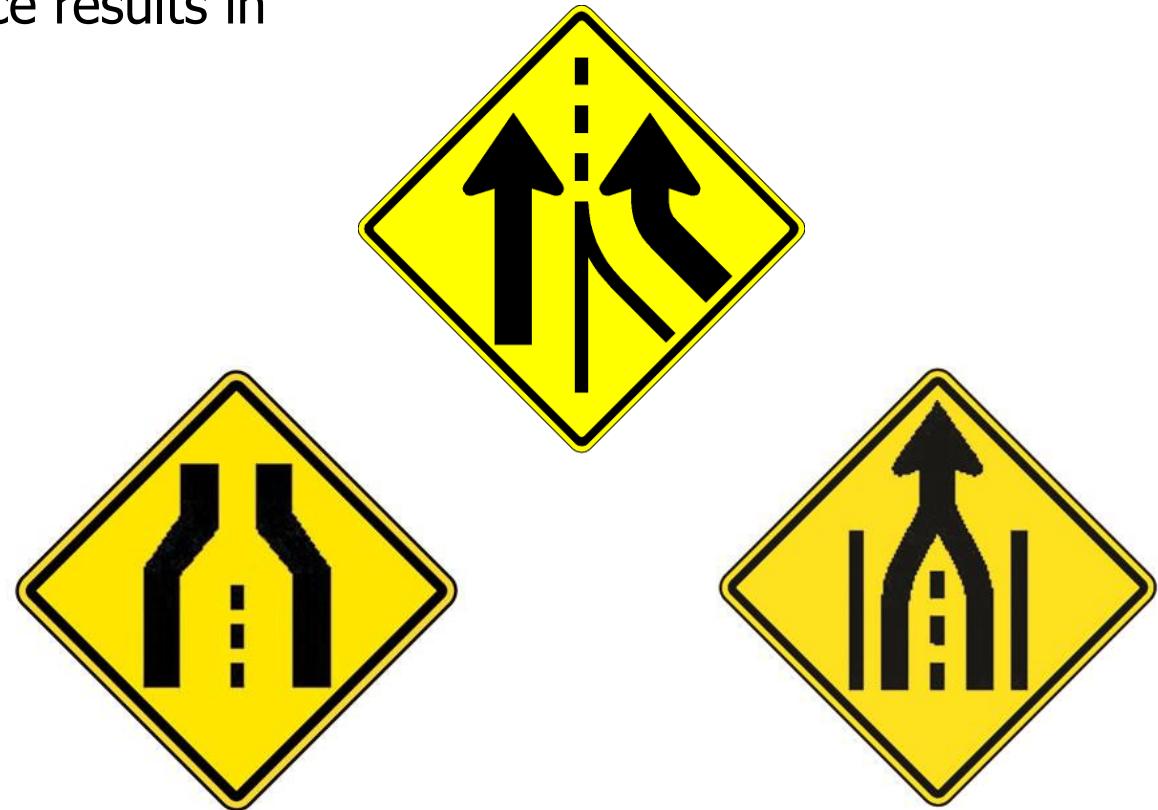


---

# Contrasting the reduce() & collect() Terminal Operations

# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways



These differences are important for parallel streams (covered later)

# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value



See [docs.oracle.com/javase/tutorial/essential/concurrency/immutability.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/immutability.html)

# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value

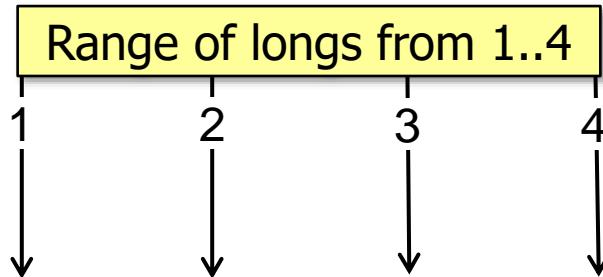
```
long factorial(long n) {
 return LongStream
 .rangeClosed(1, n)
 .reduce(1, (a, b) -> a * b);
}
```



# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value

```
long factorial(long n) {
 return LongStream
 .rangeClosed(1, n)
 .reduce(1, |(a, b) -> a * b);
}
```

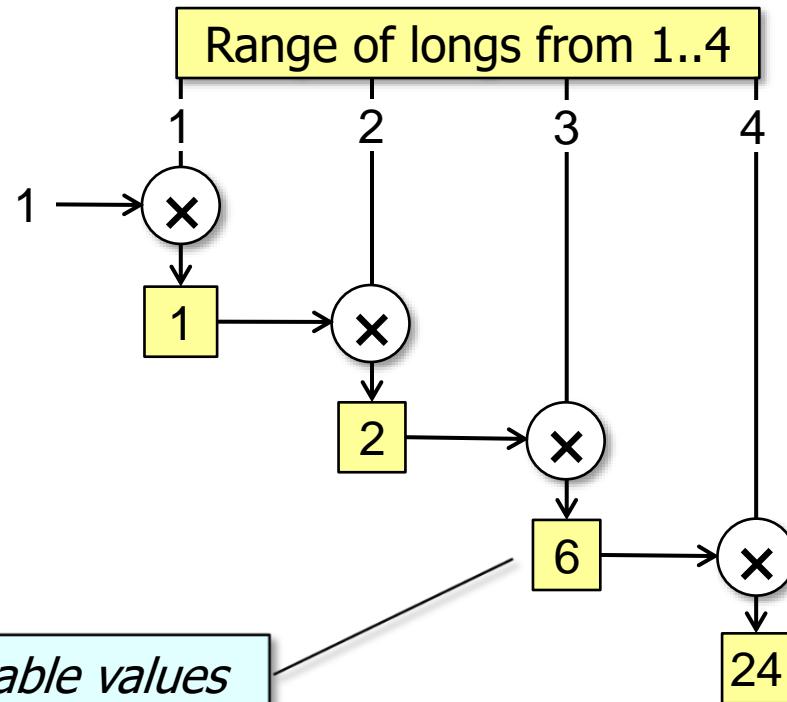


*Generate a range of primitive long values from 1 to n (inclusive)*

# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value

```
long factorial(long n) {
 return LongStream
 .rangeClosed(1, n)
 .reduce(1, (a, b) -> a * b);
}
```



*reduce() combines two immutable values (e.g., long, int, etc.) & produces a new one*

# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value
  - collect() mutates an existing value



See [greenteapress.com/thinkapjava/html/thinkjava011.html](http://greenteapress.com/thinkapjava/html/thinkjava011.html)

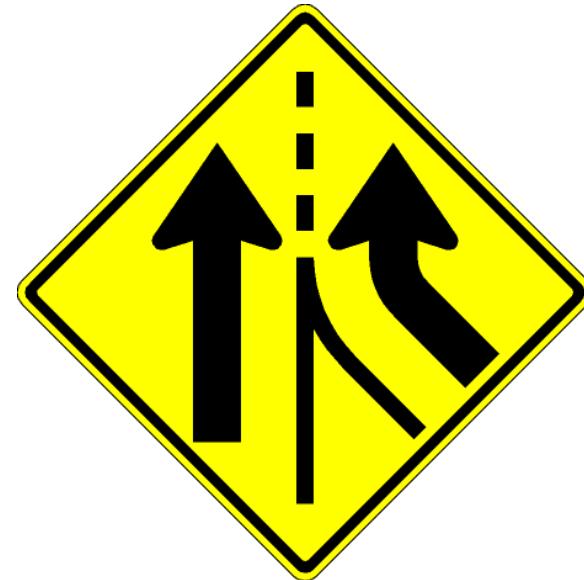
# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value
  - collect() mutates an existing value

```
Set<CharSequence> uniqueWords =
 getInput(sSHAKESPEARE),
 "\s+")
.stream()
```

```
.map(charSeq ->
 charSeq.toString()
 .toLowerCase())
```

```
.collect(toCollection(HashSet::new));
```



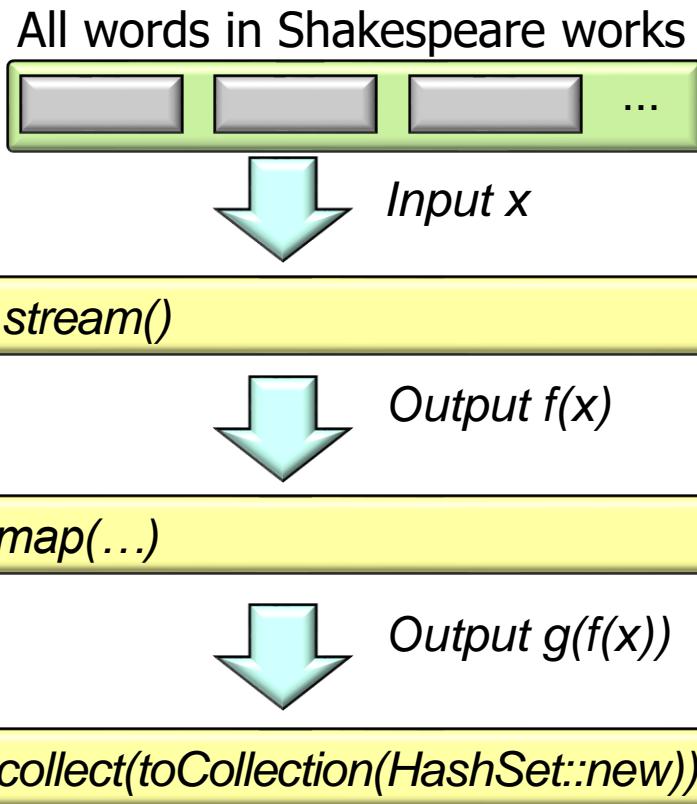
# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value
  - collect() mutates an existing value

```
Set<CharSequence> uniqueWords =
 getInput(sSHAKESPEARE),
 "\\\s+");
 .stream()
 .map(charSeq ->
 charSeq.toString()
 .toLowerCase());

.collect(toCollection(HashSet::new));
```

*Create a set of all  
the unique words in  
Shakespeare's works*



# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value
  - collect() mutates an existing value

```
Set<CharSequence> uniqueWords =
 getInput(sSHAKESPEARE),
 "\\\s+")
 .stream()
 .map(charSeq ->
 charSeq.toString()
 .toLowerCase())

.collect(toCollection(HashSet::new));
```

*Get list of all words  
in Shakespeare*

All words in Shakespeare works



*Input x*

*stream()*

*Output f(x)*

*map(...)*

*Output g(f(x))*

*collect(toCollection(HashSet::new))*

# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value
  - collect() mutates an existing value

```
Set<CharSequence> uniqueWords =
 getInput(sSHAKESPEARE),
 "\s+")
 .stream()
 .map(charSeq ->
 charSeq.toString()
 .toLowerCase())

.collect(toCollection(HashSet::new));
```

*Convert list  
into stream*

All words in Shakespeare works



*stream()*

*Input x*

*Output f(x)*

*map(...)*

*Output g(f(x))*

*collect(toCollection(HashSet::new))*

# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value
  - collect() mutates an existing value

```
Set<CharSequence> uniqueWords =
 getInput(sSHAKESPEARE),
 "\s+")
```

```
.stream()
```

*Lower case all words*

```
.map (charSeq ->
 charSeq.toString()
 .toLowerCase())
```

```
.collect(toCollection(HashSet::new)) ;
```

All words in Shakespeare works



*Input x*

*stream()*

*Output f(x)*

*map(...)*

*Output g(f(x))*

*collect(toCollection(HashSet::new))*

# Contrasting the reduce() & collect() Terminal Operations

- Terminal operations produce results in different ways, e.g.
  - reduce() creates an immutable value
  - collect() mutates an existing value

```
Set<CharSequence> uniqueWords =
 getInput(sSHAKESPEARE),
 "\\\s+")
.stream()

 Collect into a HashSet
.map (charSeq ->
 charSeq.toString()
 .toLowerCase ())

.collect (toCollection (HashSet::new)) ;
```

All words in Shakespeare works



stream()



map(...)



collect(toCollection(HashSet::new))

toCollection() creates a HashSet container & accumulates stream elements into it

---

# End of Contrast the Java Streams reduce() & collect() Terminal Operations

# Visualize WordSearcher.printResults()

Douglas C. Schmidt

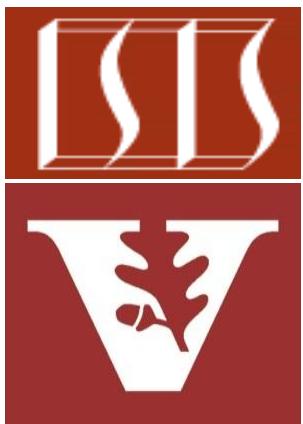
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

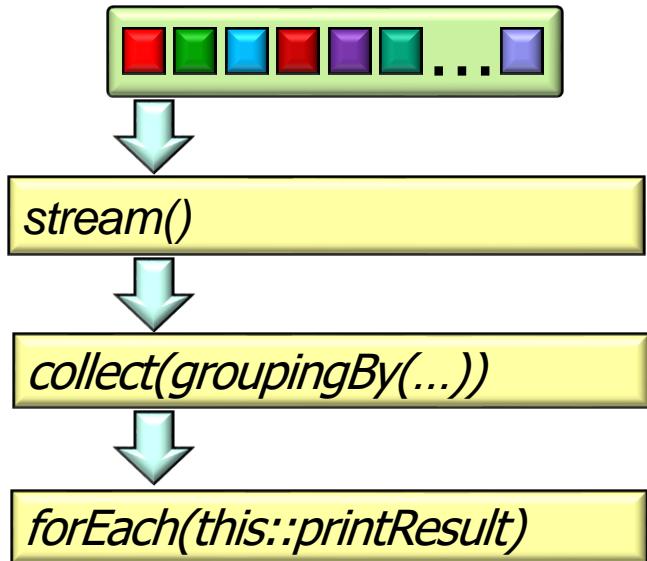
Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Visualize aggregate operations in SimpleSearchStream's WordSearcher .printResults() method



---

See [SimpleSearchStream/src/main/java/search/WordSearcher.java](#)

---

# Visualizing the Word Searcher.printResults() Method

# Visualizing the WordSearcher.printResults() Method

- Prints the index locations of any/all of the found words

```
wordSearcher.printResults(results);
```

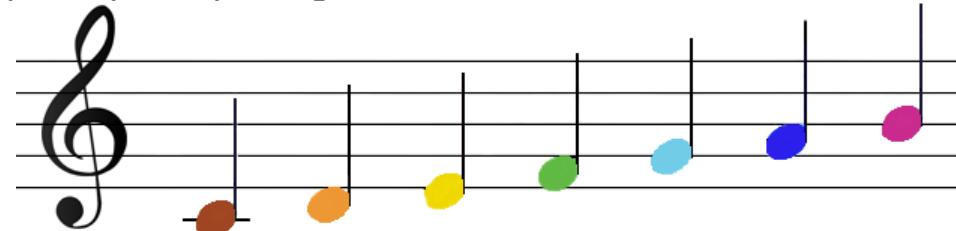
Word "Do" appeared at indices [128|138|148|199|209|219|503|972|976|997|1004|1022|1204|1216|1231|1244|1256|1266|1275|1281|1288|1297|1303|1336|1348|1358|1367|1373|1828|1832|1853|1857|1879|1919|2251|2263|2273|2302|2309] with max index of 2309

Word "Re" appeared at indices

[131|141|151|202|212|222|979|1025|1219|1259|1278|1300|1351|1370|1835|1875|1899|1939|2266|2295] with max index of 2295

Word "Mi" appeared at indices[134|144|154|205|215|225|982|1028|1213|1253|1345|1838|1872|1882|1885|1889|1922|1925|1929|2260|2292] with max index of 2292

...



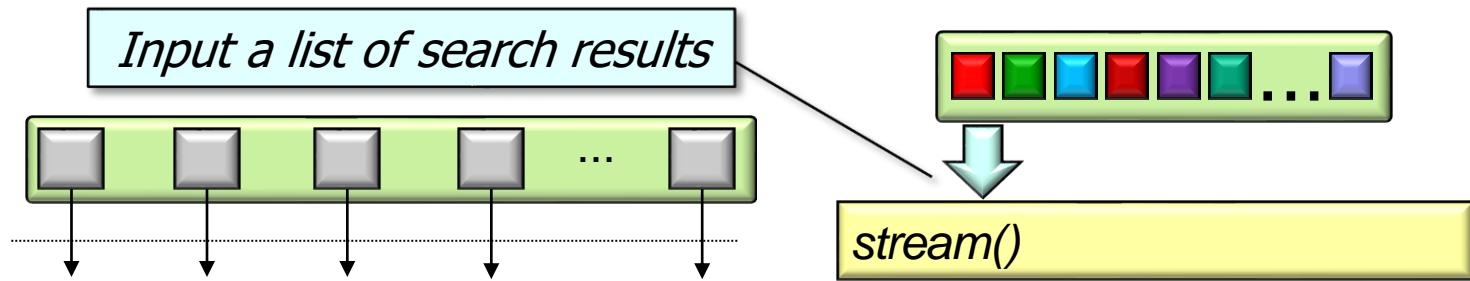
**Do Re Mi Fa Sol La Si**

This method shows the collect(groupingBy()) terminal operation

# Visualizing the WordSearcher.printResults() Method

- Prints the index locations of any/all of the found words

List  
<SearchResults>

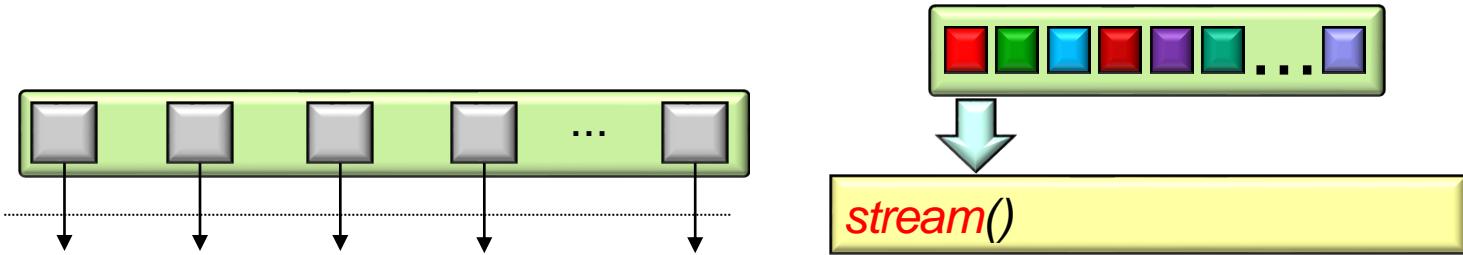


The search results correspond to words to find: "do", "re", "mi", "fa", "so", "la", "ti".

# Visualizing the WordSearcher.printResults() Method

- Prints the index locations of any/all of the found words

List  
<SearchResults>



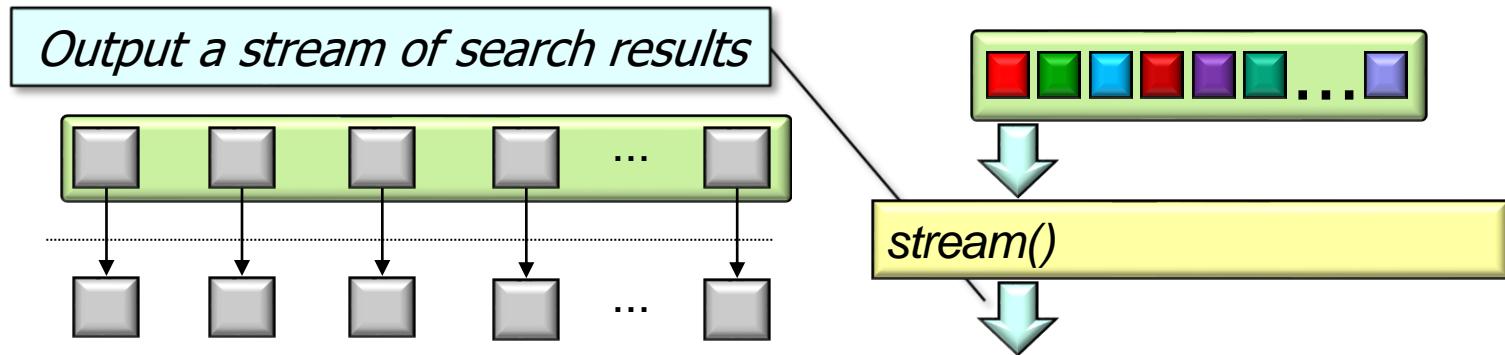
Convert list to a (sequential) stream of search results

# Visualizing the WordSearcher.printResults() Method

- Prints the index locations of any/all of the found words

List  
<SearchResults>

Stream  
<SearchResults>

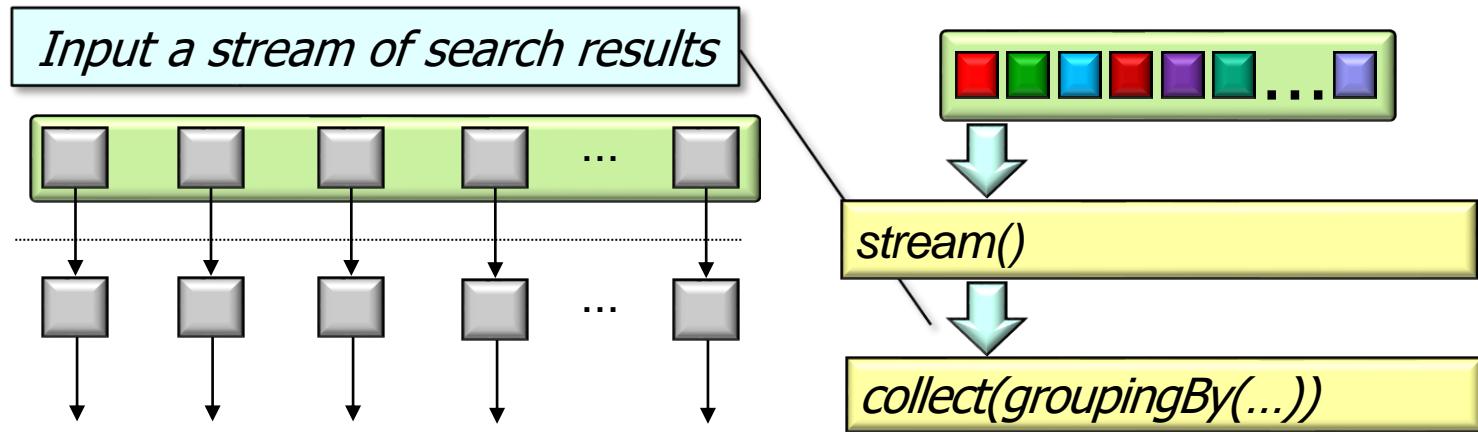


# Visualizing the WordSearcher.printResults() Method

- Prints the index locations of any/all of the found words

List  
<SearchResults>

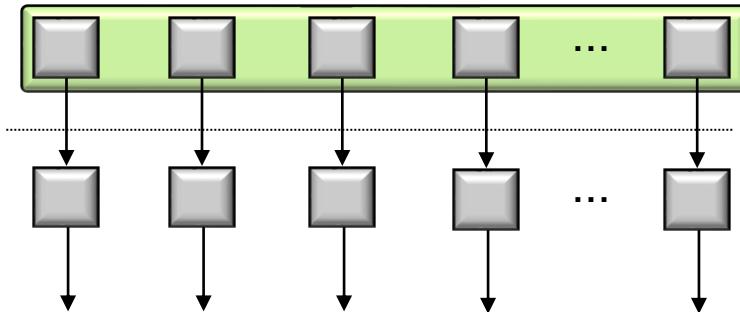
Stream  
<SearchResults>



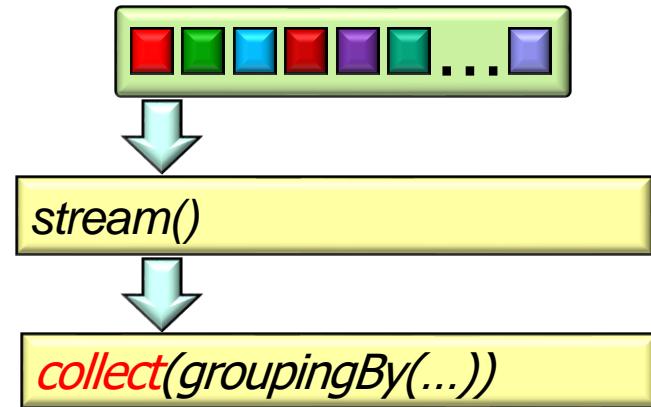
# Visualizing the WordSearcher.printResults() Method

- Prints the index locations of any/all of the found words

List  
<SearchResults>



Stream  
<SearchResults>



Trigger “intermediate operation” processing

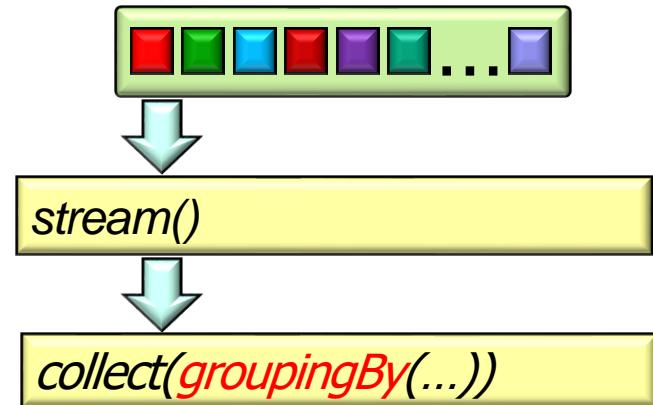
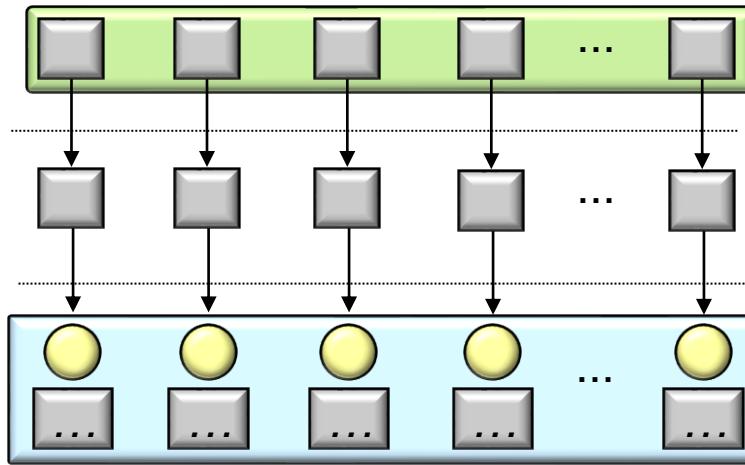
# Visualizing the WordSearcher.printResults() Method

- Prints the index locations of any/all of the found words

List  
<SearchResults>

Stream  
<SearchResults>

Map<String, List  
<Result>>



Create a map that groups words with the indices where they were found

# Visualizing the WordSearcher.printResults() Method

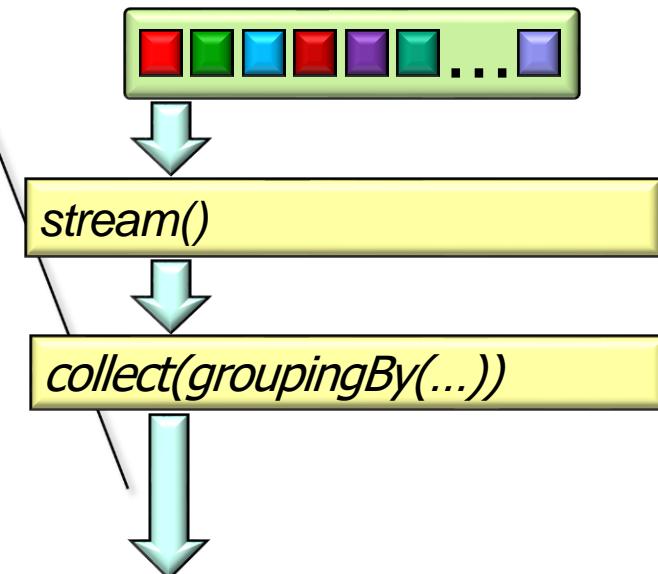
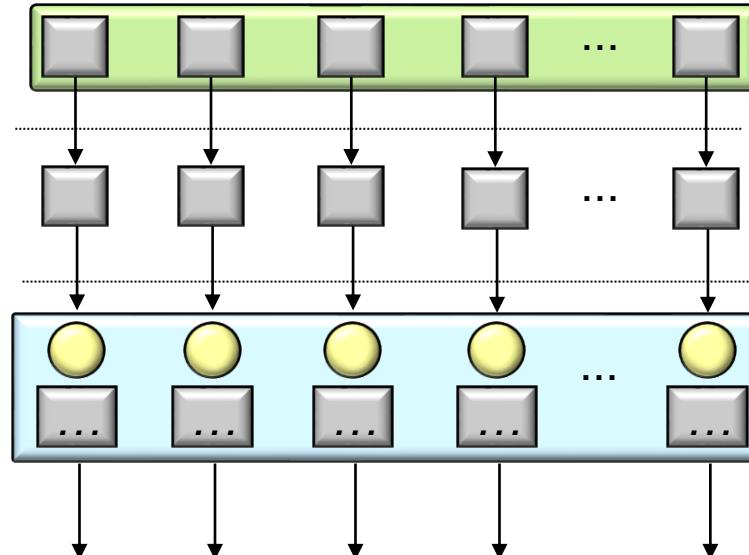
- Prints the index locations of any/all of the found words

*Output a map of strings & lists of search results*

List  
<SearchResults>

Stream  
<SearchResults>

Map<String, List  
<Result>>



# Visualizing the WordSearcher.printResults() Method

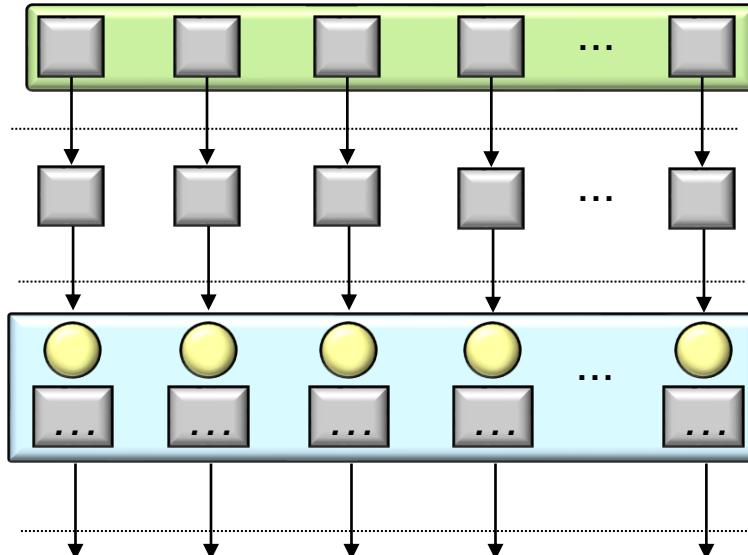
- Prints the index locations of any/all of the found words

*Input a map of strings & lists of search results*

List  
<SearchResults>

Stream  
<SearchResults>

Map<String, List  
<Result>>



*stream()*

*collect(groupingBy(...))*

*forEach(this::printResult)*

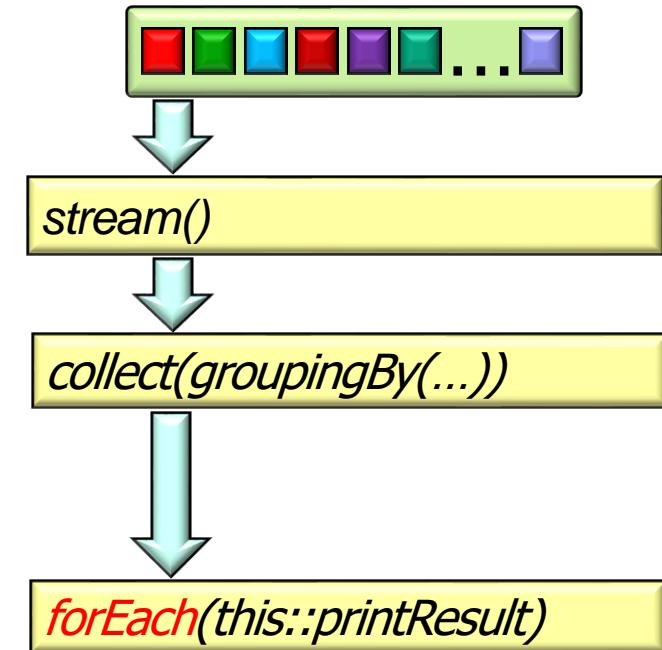
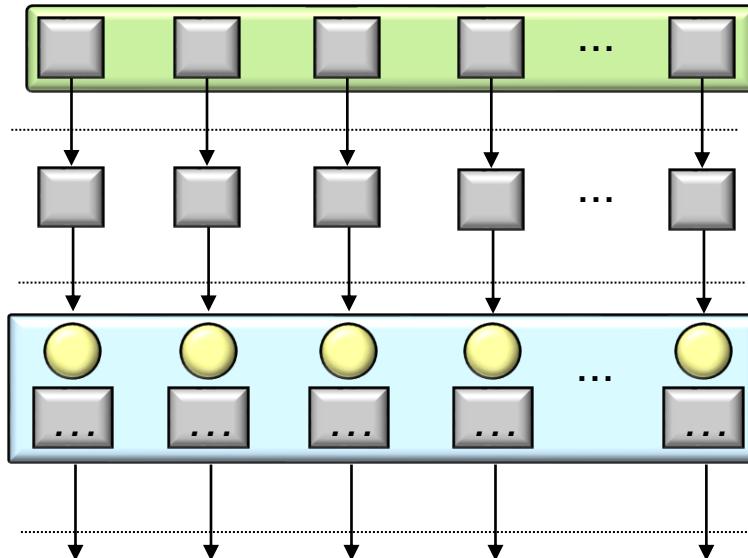
# Visualizing the WordSearcher.printResults() Method

- Prints the index locations of any/all of the found words

List  
<SearchResults>

Stream  
<SearchResults>

Map<String, List  
<Result>>



Iterate through all the key/value pairs in the map

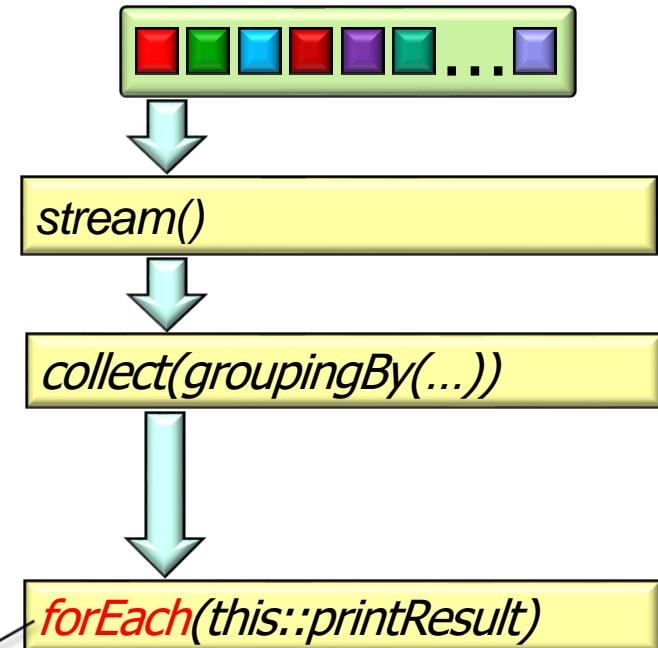
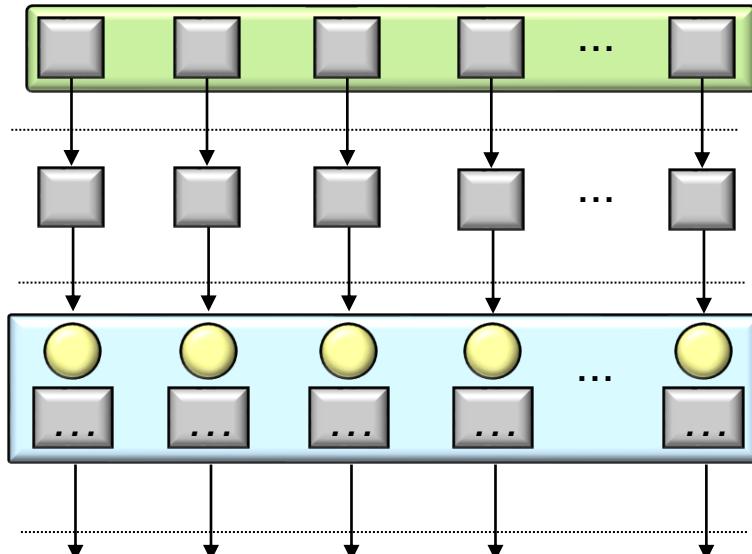
# Visualizing the WordSearcher.printResults() Method

- Prints the index locations of any/all of the found words

List  
<SearchResults>

Stream  
<SearchResults>

Map<String, List  
<Result>>



*This is the Map forEach() method, not the Stream forEach() method*

See [docs.oracle.com/javase/8/docs/api/java/util/Map.html#forEach](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#forEach)

# Visualizing the WordSearcher.printResults() Method

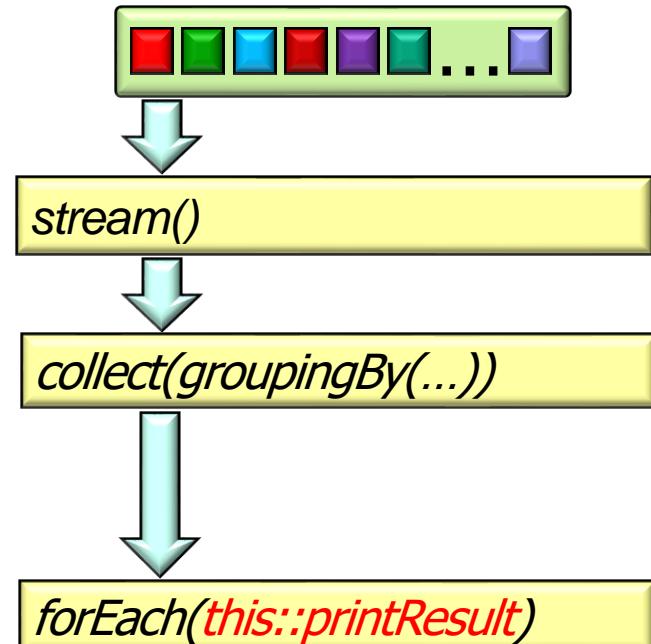
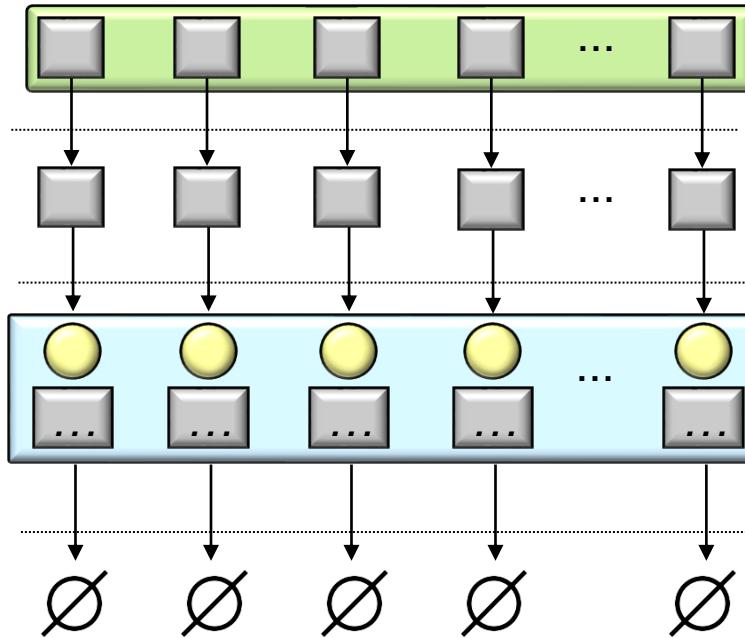
- Prints the index locations of any/all of the found words

List  
<SearchResults>

Stream  
<SearchResults>

Map<String, List  
<Result>>

Void



Print out results of each map entry (key = word & value = list of search results)

---

End of Visualize Word  
Searcher.printResults()

# Learn How to Implement **WordSearcher.printResults()**

Douglas C. Schmidt

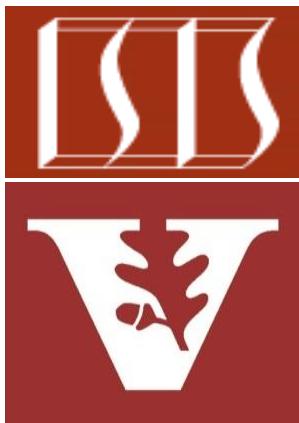
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Visualize aggregate operations in SimpleSearchStream's WordSearcher .printResults() method
- Understand the implementation of aggregate operations in SimpleSearch Stream's WordSearcher.printResults() method

```
public void printResults(List<SearchResults> listOfResults) {
 listOfResults
 .stream()

 .collect(groupingBy(SearchResults::getWord,
 LinkedHashMap::new,
 toDownstreamCollector()))

 .forEach(this::printResult);
}
```

This lesson shows the collect(groupingBy()) & mapToInt() aggregate operations

---

# Implementing the Word Searcher.printResults() Method

# Implementing the WordSearcher.printResults() Method

---

- This method prints the results of the word search

```
public void printResults(List<SearchResults> listOfResults) {
 listOfResults
 .stream()

 .collect(groupingBy(SearchResults::getWord,
 LinkedHashMap::new,
 toDownstreamCollector()))

 .forEach(this::printResult);
}
```

---

See [SimpleSearchStream/src/main/java/search/WordSearcher.java](#)

# Implementing the WordSearcher.printResults() Method

- This method prints the results of the word search

```
public void printResults(List<SearchResults> listOfResults) {
 listOfResults
 .stream()
 .collect(groupingBy(SearchResults::getWord,
 LinkedHashMap::new,
 toDownstreamCollector()))

 .forEach(this::printResult);
}
```



*Convert the list param into a stream.*

# Implementing the WordSearcher.printResults() Method

- This method prints the results of the word search

```
public void printResults(List<SearchResults> listOfResults) {
 listOfResults
 .stream()

 .collect(groupingBy(SearchResults::getWord,
 LinkedHashMap::new,
 toDownstreamCollector()))

 .forEach(this::printResult);
}
```

*Collect SearchResults into a Map, with word as the key & the list of indices as the value.*

# Implementing the WordSearcher.printResults() Method

- This method prints the results of the word search

```
public void printResults(List<SearchResults> listOfResults) {
 listOfResults
 .stream()

 .collect(groupingBy(SearchResults::getWord,
 LinkedHashMap::new,
 toDownstreamCollector()))

 .forEach(this::printResult);
}
```

*LinkedHashMap preserves the insertion order wrt iteration.*

# Implementing the WordSearcher.printResults() Method

- This method prints the results of the word search

```
public void printResults(List<SearchResults> listOfResults) {
 listOfResults
 .stream()

 .collect(groupingBy(SearchResults::getWord,
 LinkedHashMap::new,
 toDownstreamCollector()))

 .forEach(this::printResult);
}
```

*This factory method creates a downstream collector that merges results lists together.*

# Implementing the WordSearcher.printResults() Method

- This method prints the results of the word search

```
public void printResults(List<SearchResults> listOfResults) {
 listOfResults
 .stream()

 .collect(groupingBy(SearchResults::getWord,
 LinkedHashMap::new,
 toDownstreamCollector()))

 .forEach(this::printResult);
}
```

*Print out the matching  
results in the stream.*

This is the Map forEach() method *not* the Stream forEach() method!

# Implementing the WordSearcher.printResults() Method

---

- Print a word & its list of indices to the output

```
private void printResult(String word,
 List<SearchResults.Result> results) {
 System.out.print("Word \""
 + word
 + "\" appeared at indices ");
 SearchResults.printResults(results);
 System.out.println(" with max index of "
 + computeMax(results));
}
```

# Implementing the WordSearcher.printResults() Method

- Print a word & its list of indices to the output

```
private void printResult(String word,
 List<SearchResults.Result> results) {
 System.out.print("Word \"
 + word
 + "\" | appeared at indices ");

 Print the word followed by the list of search results.

 SearchResults.printResults(results);

 System.out.println(" with max index of "
 + computeMax(results));
}
```

# Implementing the WordSearcher.printResults() Method

- Print a word & its list of indices to the output

```
private void printResult(String word,
 List<SearchResults.Result> results) {
 System.out.print("Word \""
 + word
 + "\" appeared at indices ");

 SearchResults.printResults(results);

 System.out.println(" with max index of "
 + computeMax(results));
}
```

*Compute & print the max index.*

# Implementing the WordSearcher.printResults() Method

---

- Compute the max index in the list of search results

```
private int computeMax(List<SearchResults.Result> results) {
 return results
 .stream()

 .mapToInt(SearchResults.Result::getIndex)

 .max()

 .orElse(0);
}
```

This implementation works properly even if the results are not sorted!

# Implementing the WordSearcher.printResults() Method

- Compute the max index in the list of search results

```
private int computeMax(List<SearchResults.Result> results) {
 return results
 .stream()
 .mapToInt(SearchResults.Result::getIndex)
 .max()
 .orElse(0);
}
```

*Convert the list results into a stream of results*

# Implementing the WordSearcher.printResults() Method

- Compute the max index in the list of search results

```
private int computeMax(List<SearchResults.Result> results) {
 return results
 .stream()

 .mapToInt(SearchResults.Result::getIndex)
 .max()
 .orElse(0);
}
```

*Map the stream of Result objects into a stream of int primitives.*

# Implementing the WordSearcher.printResults() Method

- Compute the max index in the list of search results

```
private int computeMax(List<SearchResults.Result> results) {
 return results
 .stream()

 .mapToInt(SearchResults.Result::getIndex)

 .max()
 }
 .orElse(0);
}
```

*Returns an OptionalInt describing the maximum element of this stream or an empty optional if this stream is empty*



# Implementing the WordSearcher.printResults() Method

- Compute the max index in the list of search results

```
private int computeMax(List<SearchResults.Result> results) {
 return results
 .stream()

 .mapToInt(SearchResults.Result::getIndex)

 .max()
 .orElse(0);
}
```

*Return the value (as an int) if present, otherwise return 0.*



See [docs.oracle.com/javase/8/docs/api/java/util/OptionalInt.html#orElse](https://docs.oracle.com/javase/8/docs/api/java/util/OptionalInt.html#orElse)

---

End of Learn How To  
Implement Word  
Searcher.printResults()

# **Understand Java Streams Internals: Splitting & Combining**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**



**Professor of Computer Science**

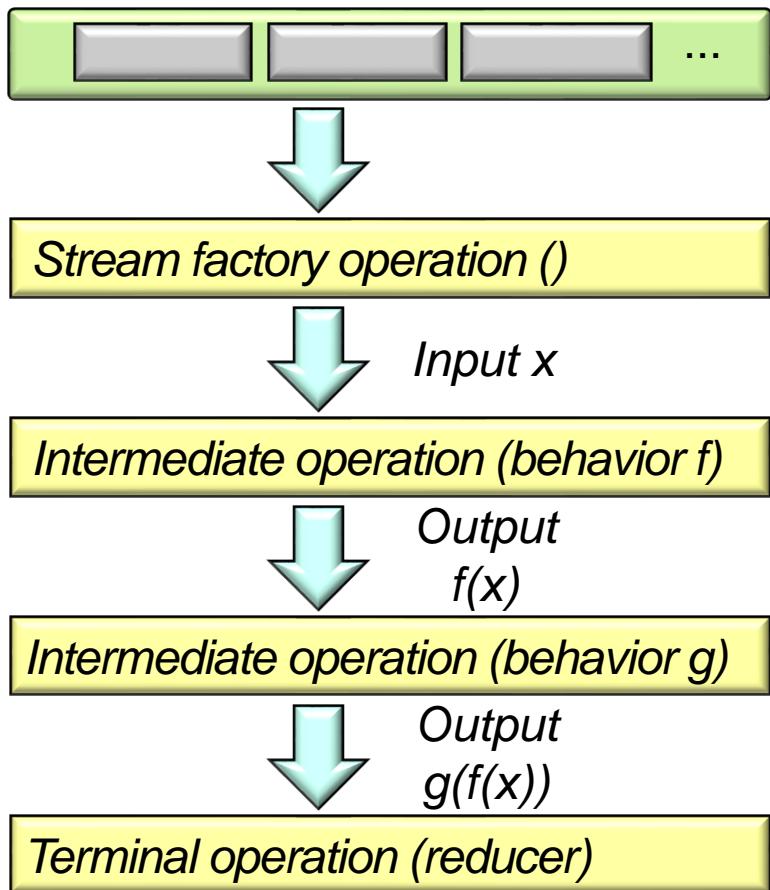
**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand stream internals



# Learning Objectives in this Part of the Lesson

---

- Understand stream internals, e.g.
  - Know what can change & what can't

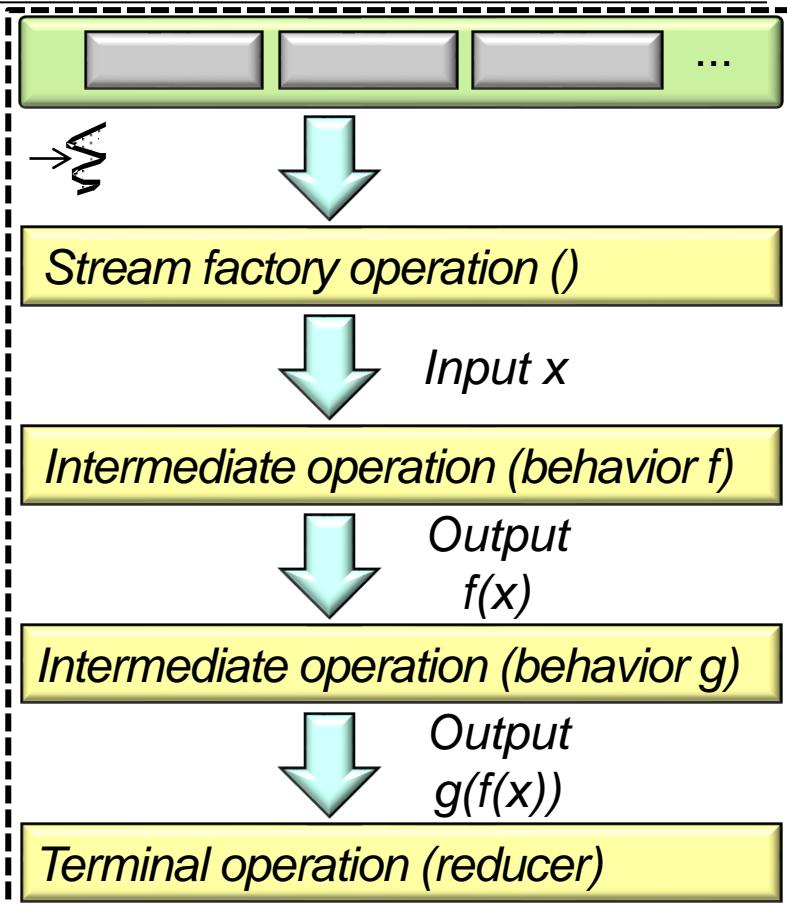
God  
Grant me the *Serenity*  
to accept the things  
I cannot change  
the *Courage* to change  
the things I can  
and the *Wisdom*  
to know the difference

---

# Why Knowledge of Streams Internals Matters

# Why Knowledge of Streams Internals Matters

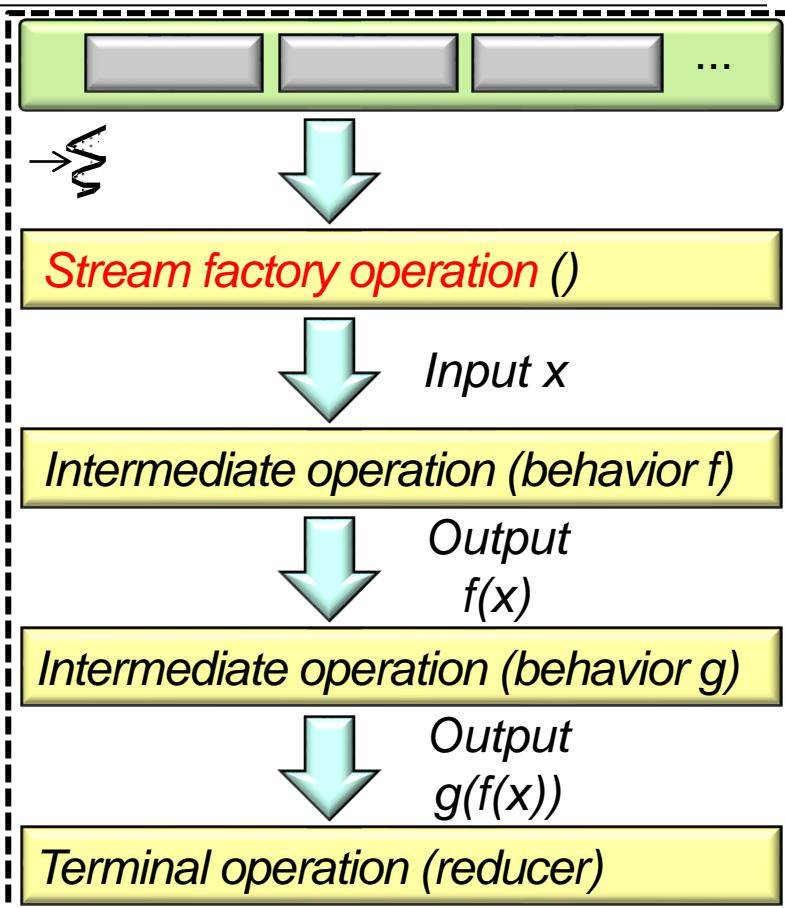
- A Java stream consists of three phases



See [www.jstatsoft.org/article/view/v040i01/v40i01.pdf](http://www.jstatsoft.org/article/view/v040i01/v40i01.pdf)

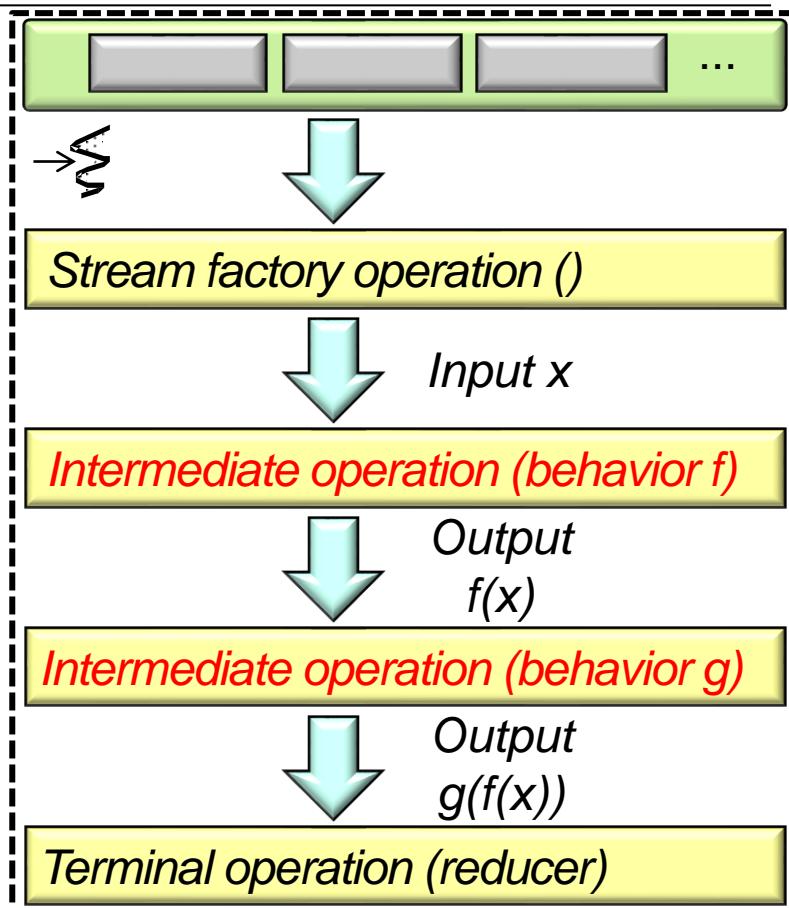
# Why Knowledge of Streams Internals Matters

- A Java stream consists of three phases
  - *Split* – Uses a spliterator to convert a data source into a stream



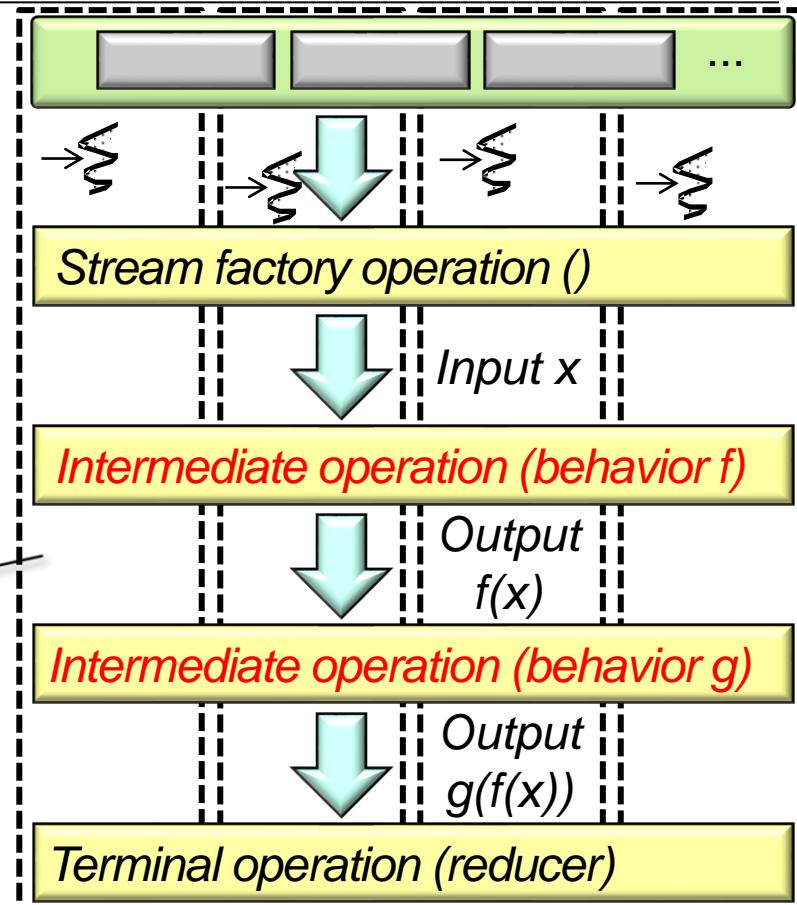
# Why Knowledge of Streams Internals Matters

- A Java stream consists of three phases
  - *Split* – Uses a spliterator to convert a data source into a stream
  - *Apply* – Process the elements in the stream



# Why Knowledge of Streams Internals Matters

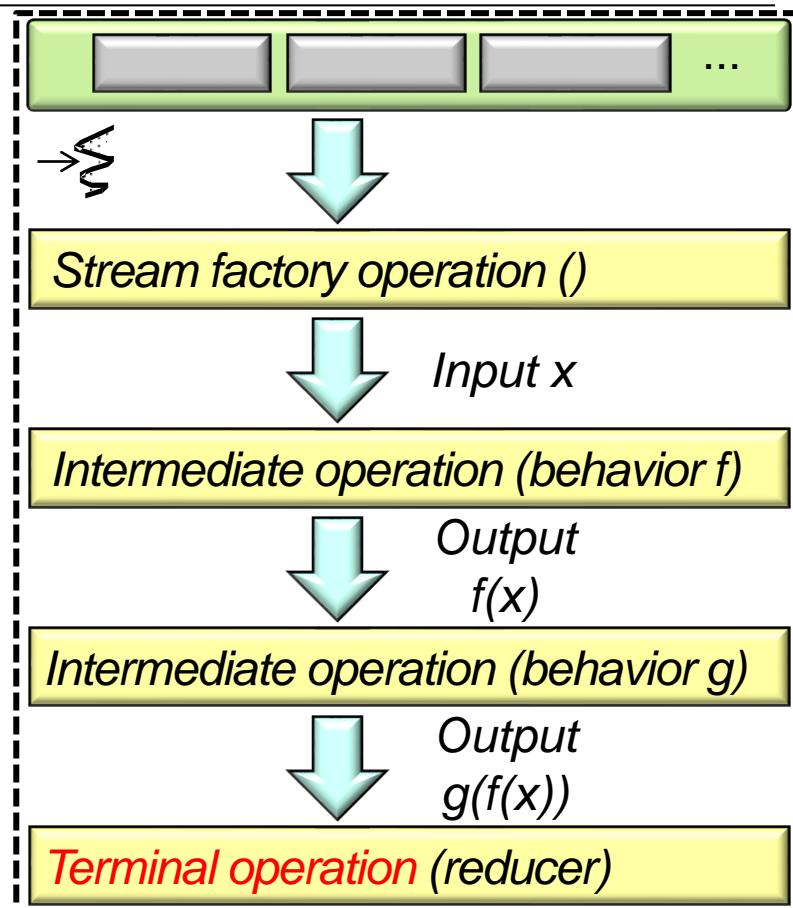
- A Java stream consists of three phases
  - *Split* – Uses a spliterator to convert a data source into a stream
  - *Apply* – Process the elements in the stream



See [docs.oracle.com/javase/tutorial/collections/stream/parallelism.html](https://docs.oracle.com/javase/tutorial/collections/stream/parallelism.html)

# Why Knowledge of Streams Internals Matters

- A Java stream consists of three phases
  - *Split* – Uses a spliterator to convert a data source into a stream
  - *Apply* – Process the elements in the stream
  - *Combine* – Trigger intermediate operation processing & create a single result



# Why Knowledge of Streams Internals Matters

- A Java stream consists of three phases
  - *Split* – Uses a spliterator to convert a data source into a stream
  - *Apply* – Process the elements in the stream
  - *Combine* – Trigger intermediate operation processing & create a single result

*Knowing which of these three phases you can control (& how to control them) is important!*

*God  
Grant me the Serenity  
to accept the things  
I cannot change  
the Courage to change  
the things I can  
and the Wisdom  
to know the difference*

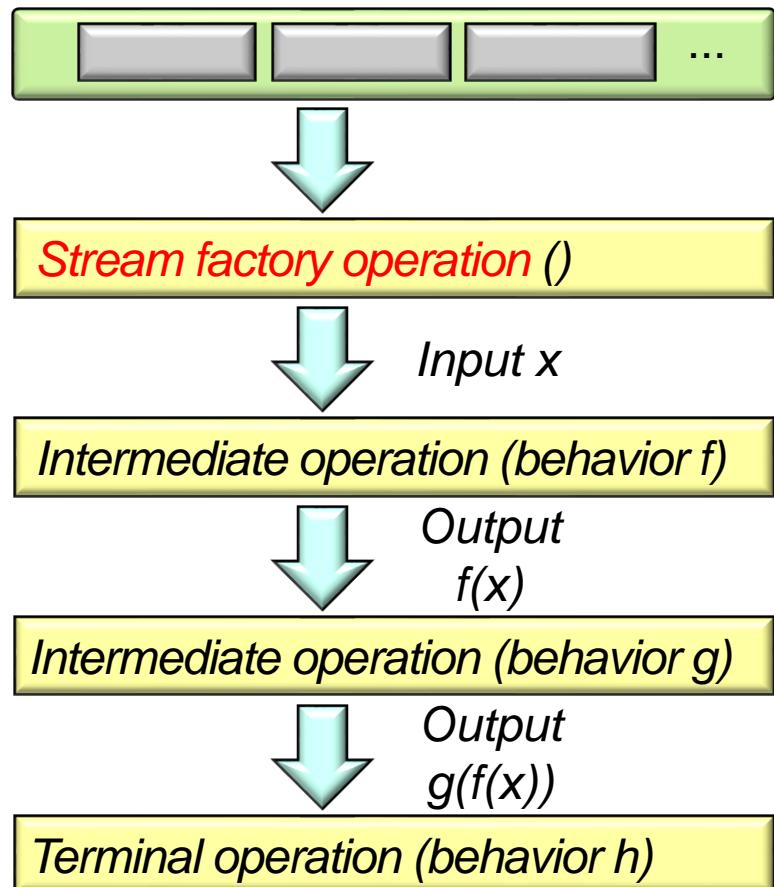
We focus on sequential stream internals now & parallel stream internals later

---

# Java Streams Splitting & Combining Mechanisms

# Java Streams Splitting & Combining Mechanisms

- A stream's splitting & combining mechanisms are often invisible

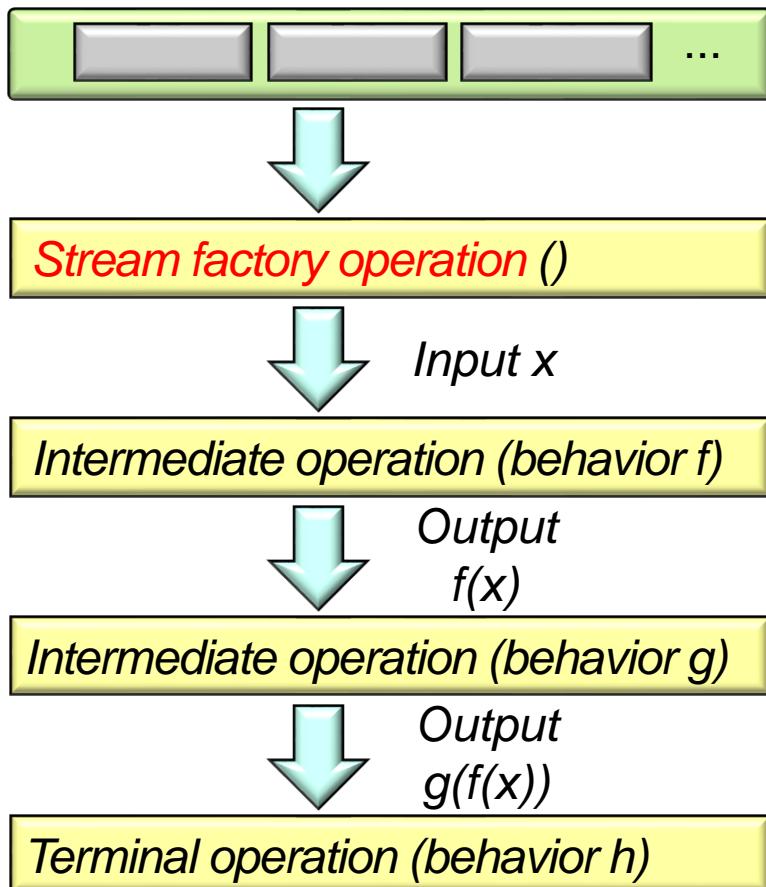


# Java Streams Splitting & Combining Mechanisms

- A stream's splitting & combining mechanisms are often invisible, e.g.
  - All Java collections have predefined spliterators

```
interface Collection<E> {
 ...
 default Spliterator<E> spliterator() {
 return Spliterators
 .spliterator(this, 0);
 }

 default Stream<E> stream() {
 return StreamSupport
 .stream(spliterator(), false);
 }
 ...
}
```



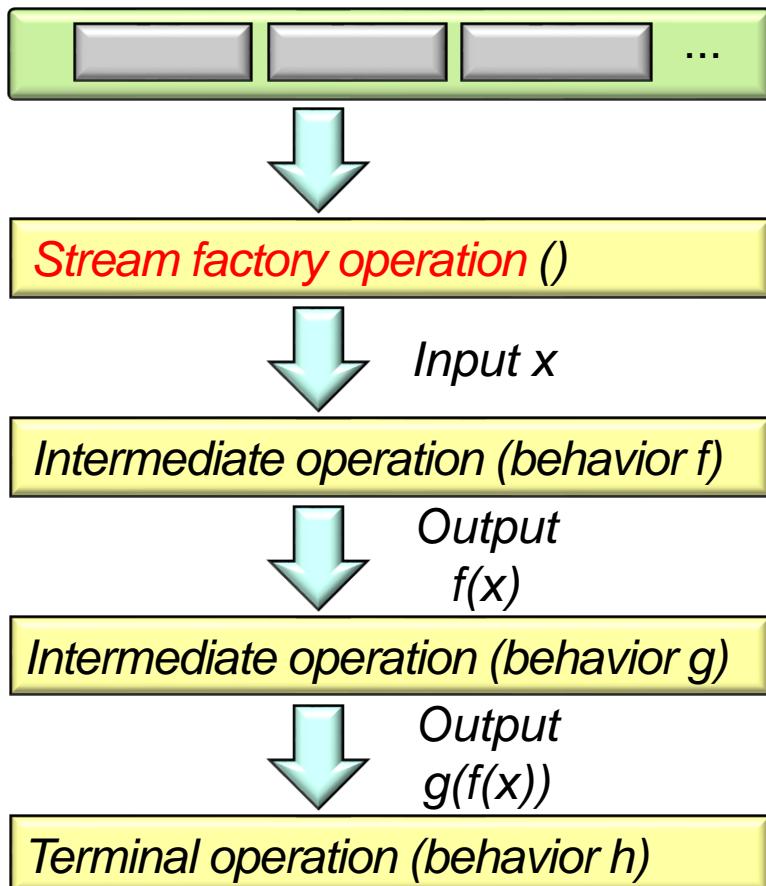
See [docs.oracle.com/javase/8/docs/api/java/util/Collection.html](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)

# Java Streams Splitting & Combining Mechanisms

- A stream's splitting & combining mechanisms are often invisible, e.g.
  - All Java collections have predefined spliterators

```
interface Collection<E> {
 ...
 default Spliterator<E> spliterator() {
 return Spliterators
 .spliterator(this, 0);
 }

 default Stream<E> stream() {
 return StreamSupport
 .stream(spliterator(), false);
 }
 ...
}
```

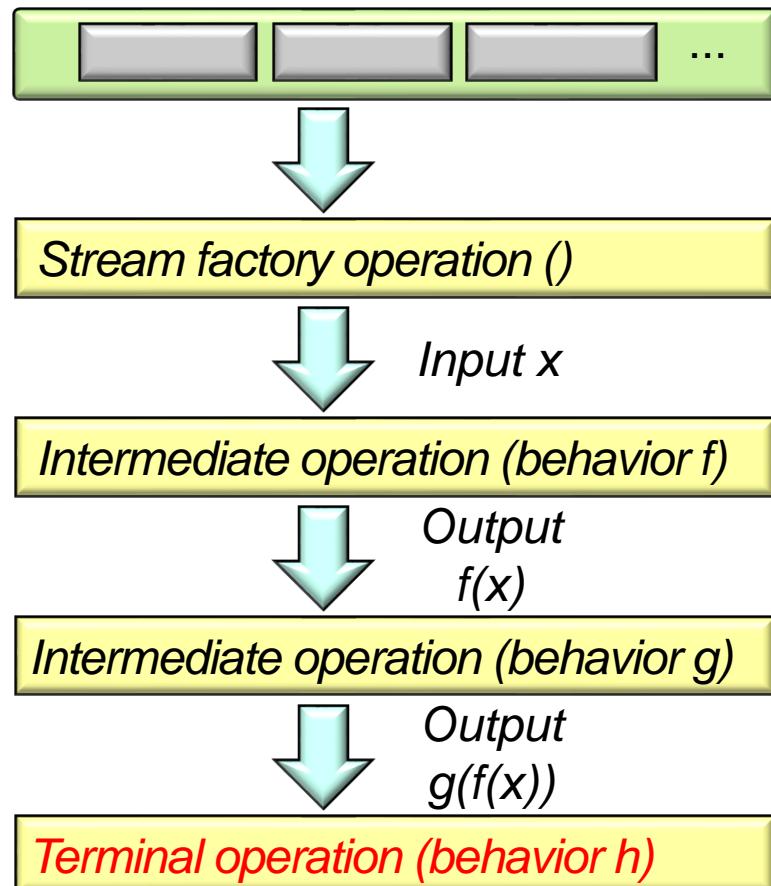


# Java Streams Splitting & Combining Mechanisms

- A stream's splitting & combining mechanisms are often invisible, e.g.
  - All Java collections have predefined spliterators
  - Java also predefines collector factory methods in the Collectors utility class

```
final class Collectors {
 ...
 public static <T> Collector<T, ?, List<T>>
 toList() { ... }

 public static <T> Collector<T, ?, Set<T>>
 toSet() { ... }
 ...
}
```

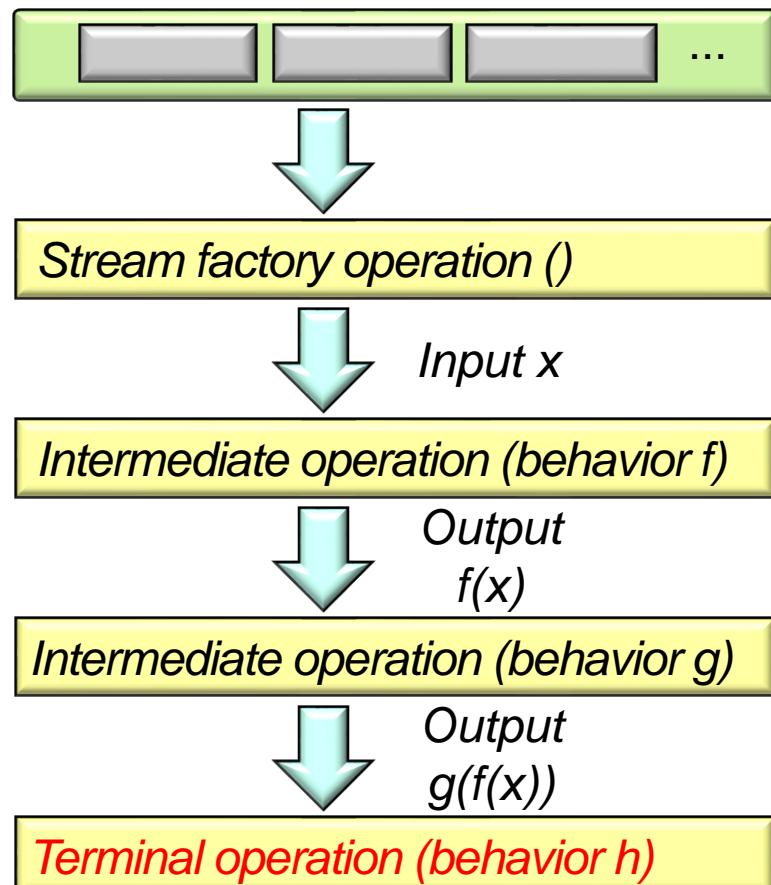


# Java Streams Splitting & Combining Mechanisms

- A stream's splitting & combining mechanisms are often invisible, e.g.
  - All Java collections have predefined spliterators
  - Java also predefines collector factory methods in the Collectors utility class

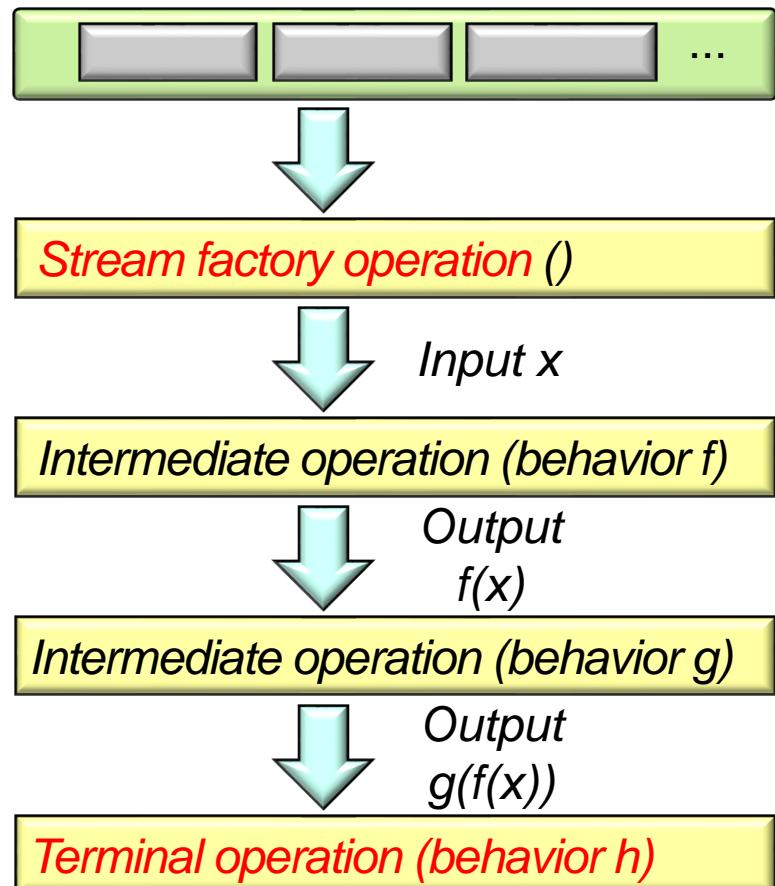
```
final class Collectors {
 ...
 public static <T> Collector<T, ?, List<T>>
 toList() { ... }

 public static <T> Collector<T, ?, Set<T>>
 toSet() { ... }
 ...
}
```



# Java Streams Splitting & Combining Mechanisms

- However, programmers can customize the behavior of splitting & combining



# Java Streams Splitting & Combining Mechanisms

- However, programmers can customize the behavior of splitting & combining



```
interface Spliterator<T> {
 boolean tryAdvance
 (Consumer<? Super T> action);

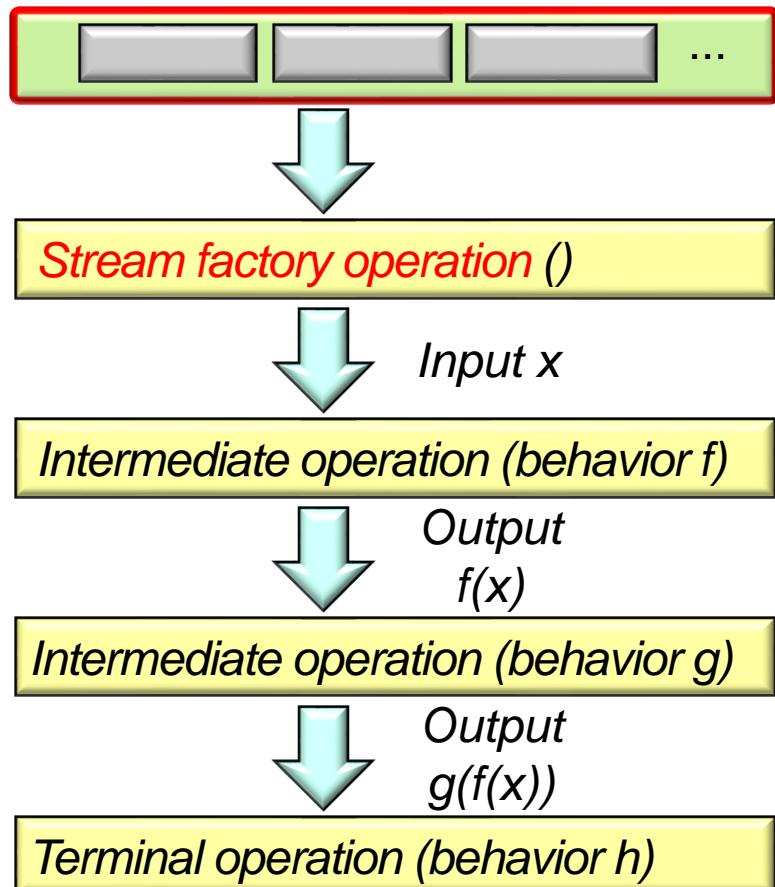
 Spliterator<T> trySplit();

 void forEachRemaining
 (Consumer<? Super T> action);

 long estimateSize();

 int characteristics();
}
```

*An interface used to traverse & partition elements of a source.*



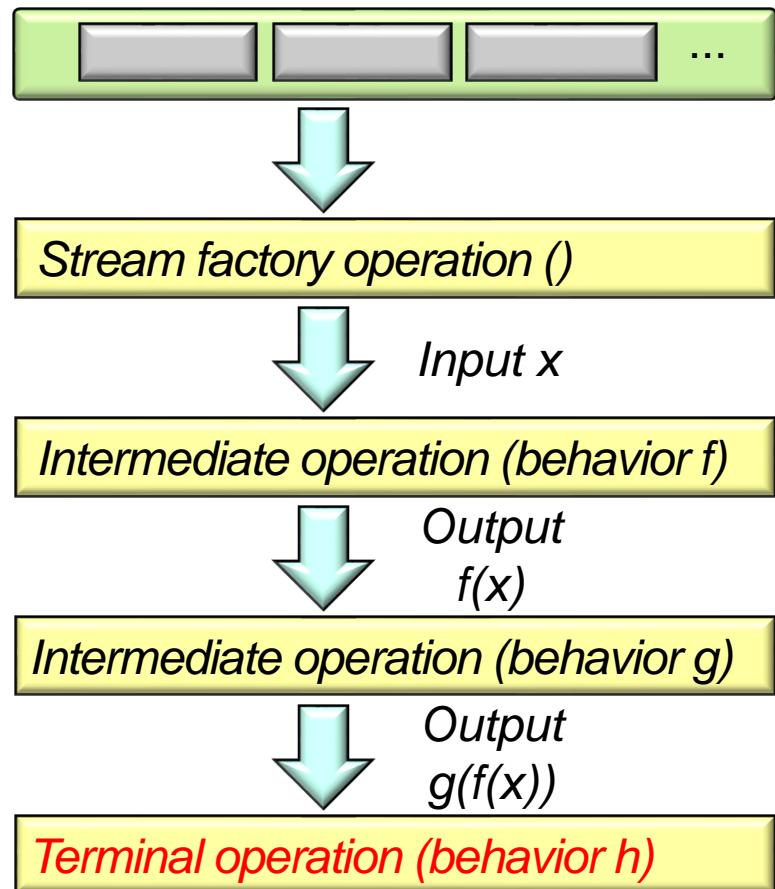
# Java Streams Splitting & Combining Mechanisms

- However, programmers can customize the behavior of splitting & combining



```
interface Collector<T,A,R> {
 Supplier<A> supplier();
 BiConsumer<A, T> accumulator();
 BinaryOperator<A> combiner();
 Function<A, R> finisher();
 Set<Collector.Characteristics>
 characteristics();
 ...
}
```

*A framework that accumulates input elements into a mutable result container.*



See [docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html)

---

# End of Understand

## Java Streams Internals: Splitting & Combining

# Understand Java Streams Spliterators

Douglas C. Schmidt

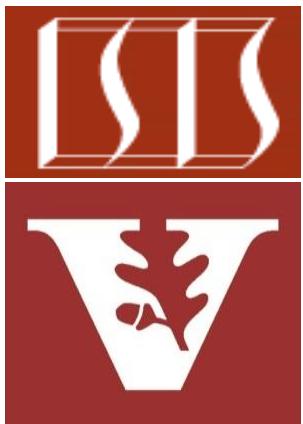
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Understand the structure & functionality of “Splittable iterators” (Spliterators)

## Interface Spliterator<T>

### Type Parameters:

T - the type of elements returned by this Spliterator

### All Known Subinterfaces:

Spliterator.OfDouble, Spliterator.OfInt, Spliterator.OfLong,  
Spliterator.OfPrimitive<T,T<sub>CONS</sub>,T<sub>SPLITR</sub>>

### All Known Implementing Classes:

Spliterators.AbstractDoubleSpliterator,  
Spliterators.AbstractIntSpliterator,  
Spliterators.AbstractLongSpliterator,  
Spliterators.AbstractSpliterator

---

### public interface Spliterator<T>

An object for traversing and partitioning elements of a source. The source of elements covered by a Spliterator could be, for example, an array, a Collection, an IO channel, or a generator function.

---

See [docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)

---

# Overview of the Java Spliterator

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8

## Interface Spliterator<T>

### Type Parameters:

T - the type of elements returned by this Spliterator

### All Known Subinterfaces:

Spliterator.OfDouble, Spliterator.OfInt, Spliterator.OfLong,  
Spliterator.OfPrimitive<T,T\_CONS,T\_SPLITR>

### All Known Implementing Classes:

Spliterators.AbstractDoubleSpliterator,  
Spliterators.AbstractIntSpliterator,  
Spliterators.AbstractLongSpliterator,  
Spliterators.AbstractSpliterator

## public interface Spliterator<T>

An object for traversing and partitioning elements of a source. The source of elements covered by a Spliterator could be, for example, an array, a Collection, an IO channel, or a generator function.

A Spliterator may traverse elements individually (`tryAdvance()`) or sequentially in bulk (`forEachRemaining()`).

See [docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)

# Overview of the Java Spliterator

---

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source



```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

for (Spliterator<String> s =
 quote.spliterator();
 s.tryAdvance(System.out::print)
 != false;
)
 continue;
```

# Overview of the Java Spliterator

---

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
  - e.g., a collection, array, etc.

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 . . .);

for (Spliterator<String> s =
 quote.spliterator();
 s.tryAdvance(System.out::print)
 != false;
)
 continue;
```

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
  - e.g., a collection, array, etc.

*This source is an array/list of strings*

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

for (Spliterator<String> s =
 quote.spliterator();
 s.tryAdvance(System.out::print)
 != false;
)
 continue;
```

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
  - e.g., a collection, array, etc.

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 . . .);

for (Spliterator<String> s =
 quote.spliterator();
 s.tryAdvance(System.out::print)
 != false;
)
 continue;
```

Create a spliterator for  
the entire array/list

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
  - e.g., a collection, array, etc.

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 . . .);

for (Spliterator<String> s =
 quote.spliterator();
 s.tryAdvance(System.out::print)
 != false;
)
 continue;
```

*tryAdvance()* combines  
the *hasNext()* & *next()*  
methods of *Iterator*

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
    - e.g., a collection, array, etc.

```
boolean tryAdvance(Consumer<? super T> action) {
 if (noMoreElementsRemain)
 return false;
 else { ...
 action.accept
 (nextElement);
 return true;
 }
}
```

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

for (Spliterator<String> s =
 quote.spliterator();
 s.tryAdvance(System.out::print)
 != false;
)
 continue;
```

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
    - e.g., a collection, array, etc.

```
boolean tryAdvance(Consumer<? super T> action) {
 if (noMoreElementsRemain)
 return false;
 else { ...
 action.accept
 (nextElement);
 return true;
 }
}
```

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

for (Spliterator<String> s =
 quote.spliterator();
 s.tryAdvance(System.out::print)
 != false;
)
 continue;
```

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
    - e.g., a collection, array, etc.

```
boolean tryAdvance(Consumer<? super T> action) {
 if (noMoreElementsRemain)
 return false;
 else { ...
 action.accept
 (nextElement);
 return true;
 }
}
```

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

for (Spliterator<String> s =
 quote.spliterator();
 s.tryAdvance(System.out::print)
 != false;
)
 continue;
```

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
  - e.g., a collection, array, etc.

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

for (Spliterator<String> s =
 quote.spliterator();
 s.tryAdvance(System.out::print)
 != false;
)
 continue;
```

*Print value of each string in the quote*

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
  - *Split* – It can also partition all elements of a source



```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 . . .);

Spliterator<String> secondHalf =
 quotespliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();

firstHalf.forEachRemaining
 (System.out::print);
secondHalf.forEachRemaining
 (System.out::print);
```

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
  - *Split* – It can also partition all elements of a source

Create a spliterator for the entire array/list

```
List<String> quote = List.of("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

Spliterator<String> secondHalf =
 quote.spliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();

firstHalf.forEachRemaining
 (System.out::print);
secondHalf.forEachRemaining
 (System.out::print);
```

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - Iterator* – It can be used to traverse elements of a source
  - Split* – It can also partition all elements of a source

*trySplit() returns a spliterator covering elements that will no longer be covered by the invoking spliterator*

```
List<String> quote = List.of("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

Spliterator<String> secondHalf =
 quote.spliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();

firstHalf.forEachRemaining
 (System.out::print);
secondHalf.forEachRemaining
 (System.out::print);
```

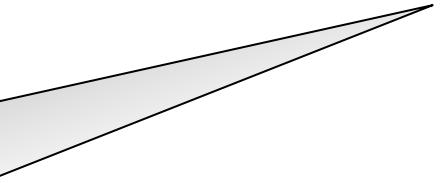
# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - Iterator* – It can be used to traverse elements of a source
  - Split* – It can also partition all elements of a source

```
Spliterator<T> trySplit() {
 if (input <= minimum size)
 return null
 else {
 split input in 2 chunks
 update "right chunk"
 return spliterator
 for "left chunk"
 }
}
```

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

Spliterator<String> secondHalf =
 quote.spliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();
```



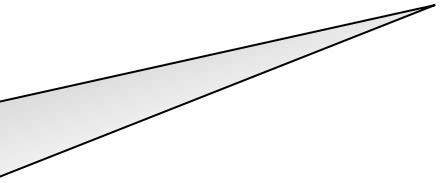
# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - Iterator* – It can be used to traverse elements of a source
  - Split* – It can also partition all elements of a source

```
Spliterator<T> trySplit() {
 if (input <= minimum size)
 return null
 else {
 split input in 2 chunks
 update "right chunk"
 return spliterator
 for "left chunk"
 }
}
```

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

Spliterator<String> secondHalf =
 quote.spliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();
```



trySplit() calls itself recursively until all chunks are  $\leq$  to the minimize size

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - Iterator* – It can be used to traverse elements of a source
  - Split* – It can also partition all elements of a source

```
Spliterator<T> trySplit() {
 if (input <= minimum size)
 return null
 else {
 split input in 2 chunks
 update "right chunk"
 return spliterator
 for "left chunk"
 }
}
```

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);
```

```
Spliterator<String> secondHalf =
 quote.spliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();
```



Ideally, a spliterator efficiently splits the original input source in half!

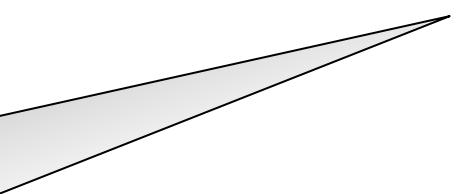
# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - Iterator* – It can be used to traverse elements of a source
  - Split* – It can also partition all elements of a source

```
Spliterator<T> trySplit() {
 if (input <= minimum size)
 return null
 else {
 split input in 2 chunks
 update "right chunk"
 return spliterator
 for "left chunk"
 }
}
```

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

Spliterator<String> secondHalf =
 quote.spliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();
```



The “right chunk” is defined by updating the state of **this** spliterator object

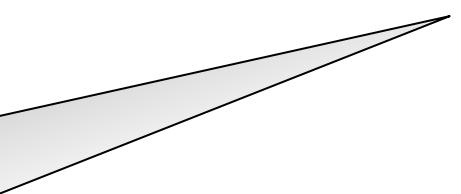
# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - Iterator* – It can be used to traverse elements of a source
  - Split* – It can also partition all elements of a source

```
Spliterator<T> trySplit() {
 if (input <= minimum size)
 return null
 else {
 split input in 2 chunks
 update "right chunk"
 return spliterator
 for "left chunk"
 }
}
```

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

Spliterator<String> secondHalf =
 quote.spliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();
```



The “left chunk” is defined by creating/returning a new spliterator object

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - Iterator* – It can be used to traverse elements of a source
  - Split* – It can also partition all elements of a source

*Performs the action for each element in the spliterator*

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);

Spliterator<String> secondHalf =
 quote.spliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();

firstHalf.forEachRemaining
 (System.out::print);
secondHalf.forEachRemaining
 (System.out::print);
```

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - Iterator* – It can be used to traverse elements of a source
  - Split* – It can also partition all elements of a source

```
List<String> quote = List.of
 ("This ", "above ", "all- ",
 "to ", "thine ", "own ",
 "self ", "be ", "true", "\n",
 ...);
```

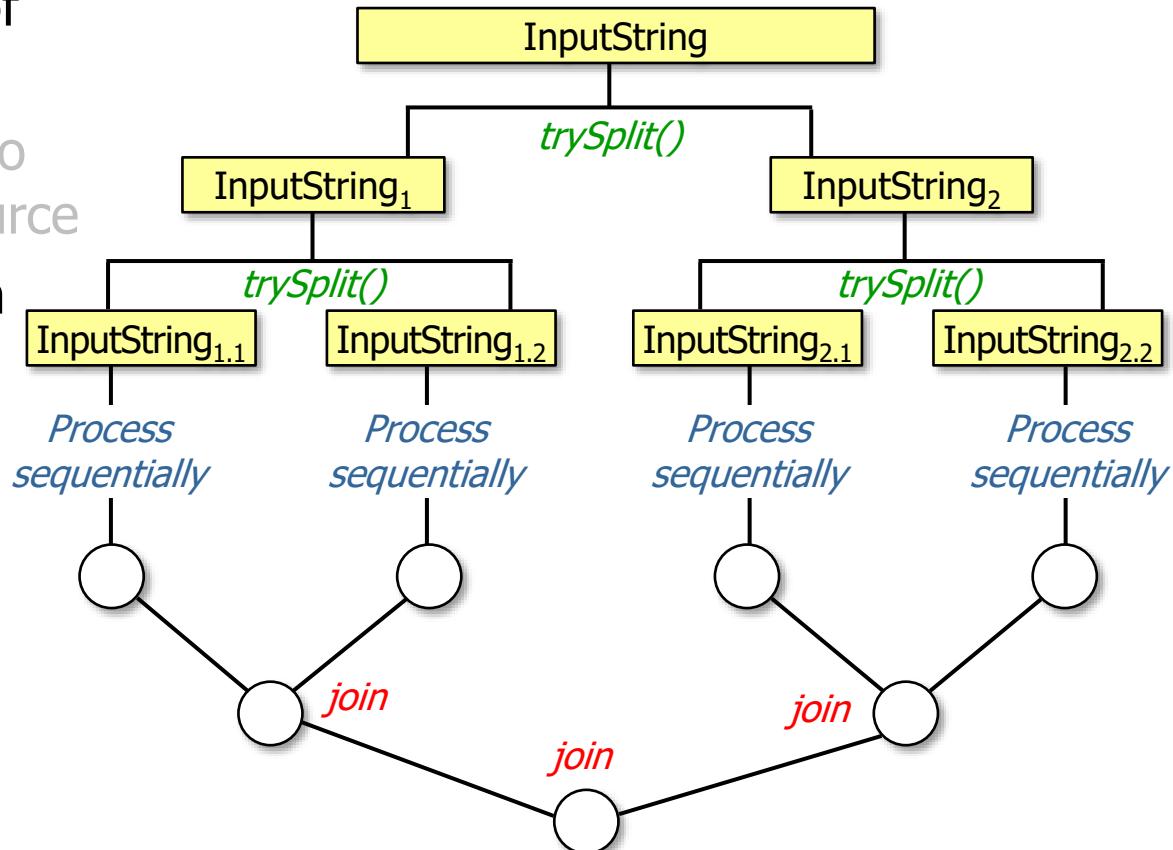
```
Spliterator<String> secondHalf =
 quotespliterator();
Spliterator<String> firstHalf =
 secondHalf.trySplit();
```

*Print value of each string in the quote*

```
firstHalf.forEachRemaining
 (System.out::print);
secondHalf.forEachRemaining
 (System.out::print);
```

# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
  - *Split* – It can also partition all elements of a source
    - Mostly used with Java parallel streams



# Overview of the Java Spliterator

- A Spliterator is a new type of "splittable iterator" in Java 8
  - *Iterator* – It can be used to traverse elements of a source
  - *Split* – It can also partition all elements of a source



## Interface Spliterator<T>

### Type Parameters:

T - the type of elements returned by this Spliterator

### All Known Subinterfaces:

Spliterator.OfDouble, Spliterator.OfInt, Spliterator.OfLong,  
Spliterator.OfPrimitive<T,T\_CONS,T\_SPLITR>

### All Known Implementing Classes:

Spliterators.AbstractDoubleSpliterator,  
Spliterators.AbstractIntSpliterator,  
Spliterators.AbstractLongSpliterator,  
Spliterators.AbstractSpliterator

### public interface Spliterator<T>

An object for traversing and partitioning elements of a source. The source of elements covered by a Spliterator could be, for example, an array, a Collection, an IO channel, or a generator function.

A Spliterator may traverse elements individually (`tryAdvance()`) or sequentially in bulk (`forEachRemaining()`).

We focus on traversal now & on partitioning later when covering parallel streams

# Overview of the Java Spliterator

- The StreamSupport.stream() factory method creates a new sequential or parallel stream from a Spliterator

## stream

```
public static <T> Stream<T> stream(Spliterator<T> spliterator,
 boolean parallel)
```

Creates a new sequential or parallel Stream from a Spliterator.

The spliterator is only traversed, split, or queried for estimated size after the terminal operation of the stream pipeline commences.

It is strongly recommended the spliterator report a characteristic of IMMUTABLE or CONCURRENT, or be late-binding. Otherwise, stream(java.util.function.Supplier, int, boolean) should be used to reduce the scope of potential interference with the source. See Non-Interference for more details.

**Type Parameters:**

T - the type of stream elements

**Parameters:**

spliterator - a Spliterator describing the stream elements

parallel - if true then the returned stream is a parallel stream; if false the returned stream is a sequential stream.

**Returns:**

a new sequential or parallel Stream

# Overview of the Java Spliterator

- The StreamSupport.stream() factory method creates a new sequential or parallel stream from a spliterator
  - e.g., the Collection interface defines two default methods using this capability

```
public interface Collection<E>
 extends Iterable<E> {
 ...
 default Stream<E> stream() {
 return StreamSupport
 .stream(spliterator(),
 false);
 }

 default Stream<E>
 parallelStream() {
 return StreamSupport
 .stream(spliterator(),
 true);
 }
}
```

See [jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/Collection.java](https://jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/Collection.java)

# Overview of the Java Spliterator

- The StreamSupport.stream() factory method creates a new sequential or parallel stream from a spliterator
  - e.g., the Collection interface defines two default methods using this capability

*The 'false' parameter creates a sequential stream, whereas 'true' creates a parallel stream*

```
public interface Collection<E>
 extends Iterable<E> {
 ...
 default Stream<E> stream() {
 return StreamSupport
 .stream(spliterator(),
 false);
 }
 default Stream<E>
 parallelStream() {
 return StreamSupport
 .stream(spliterator(),
 true);
 }
}
```

---

# End of Understand Java Streams Spliterators

# Apply Java Streams Spliterators

Douglas C. Schmidt

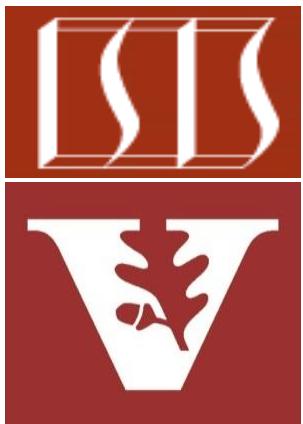
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

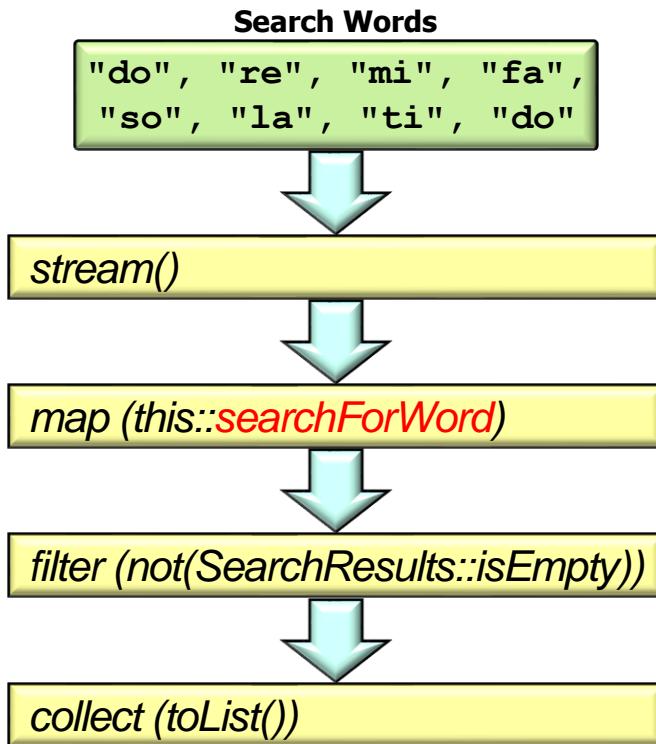
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of "Splittable iterators" (Spliterators)
- Recognize how to apply Spliterator to the SimpleSearchStream program

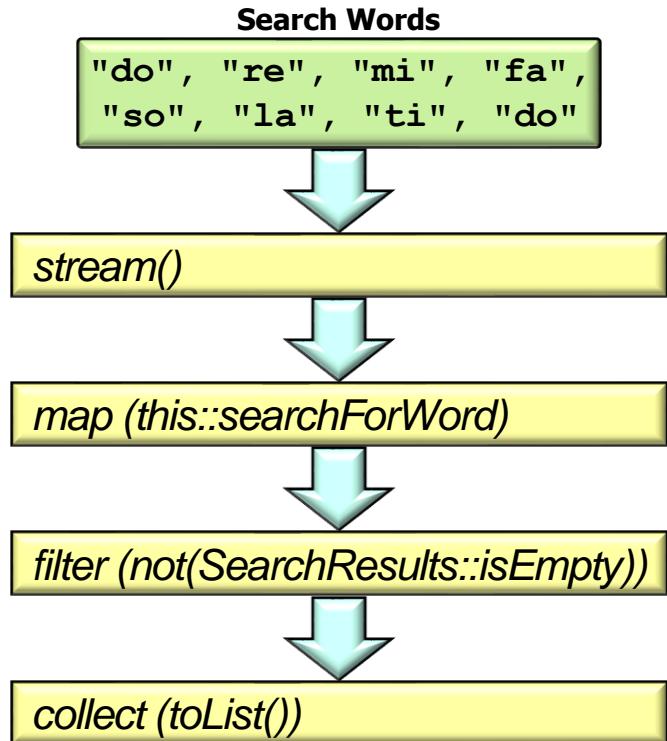


---

# Applying Java Spliterator in SimpleSearchStream

# Applying Java Spliterator in SimpleSearchStream

- The SimpleSearchStream program uses a sequential spliterator

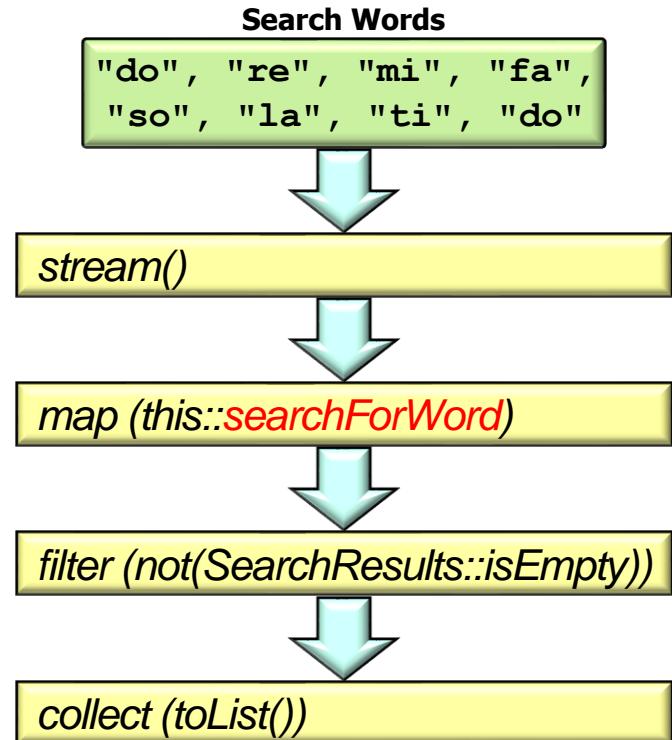


# Applying Java Spliterator in SimpleSearchStream

- searchForWord() uses the spliterator to find all instances of a word in the input & return a SearchResults object

**SearchResults searchForWord**

```
(String word) {
 return new SearchResults
 (..., word, ..., StreamSupport
 .stream(new WordMatchSpliterator
 (mInput, word),
 false)
 .collect(toList()));
}
```



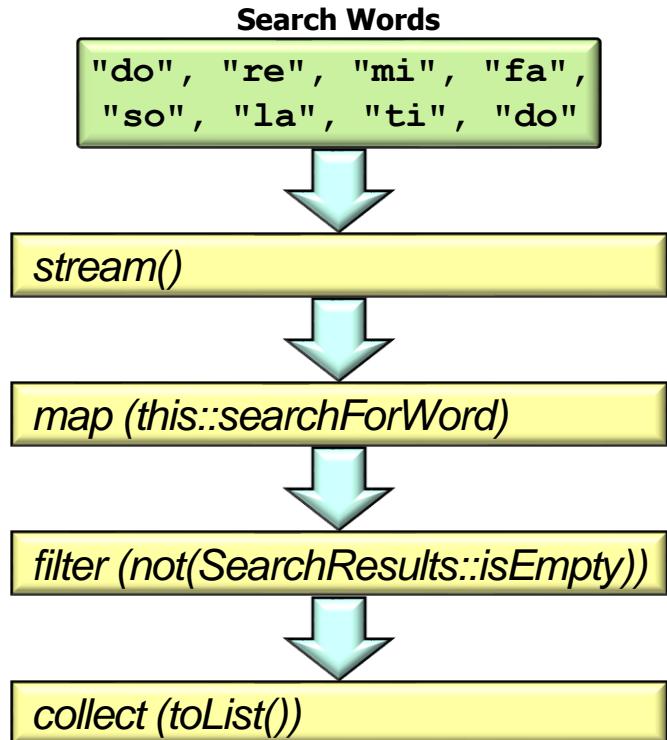
See [SimpleSearchStream/src/main/java/search/WordSearcher.java](#)

# Applying Java Spliterator in SimpleSearchStream

- searchForWord() uses the spliterator to find all instances of a word in the input & return a SearchResults object

```
SearchResults searchForWord
 (String word) {
 return new SearchResults
 (..., word, ..., StreamSupport
 .stream(new WordMatchSpliterator
 (mInput, word),
 false)
 .collect(toList()));
}
```

*StreamSupport.stream() creates a sequential stream via the WordMatchSpliterator class*

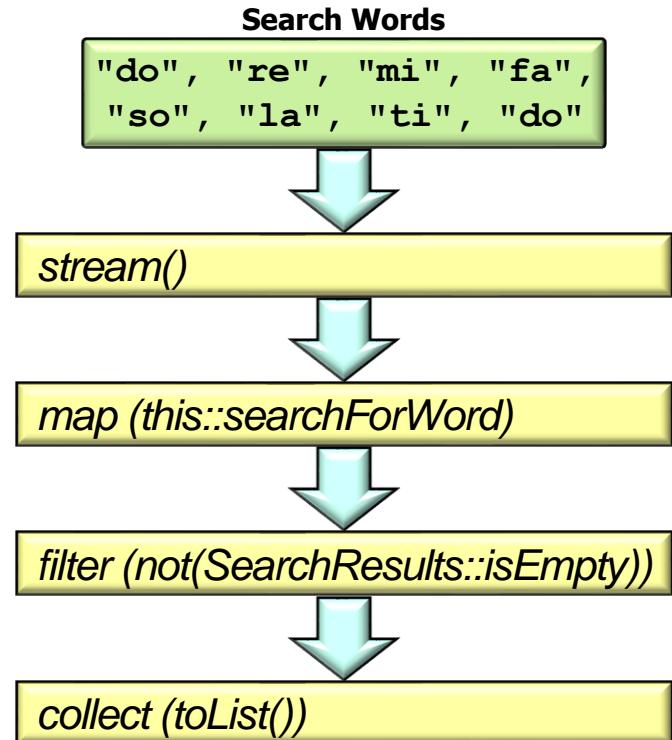


# Applying Java Spliterator in SimpleSearchStream

- searchForWord() uses the spliterator to find all instances of a word in the input & return a SearchResults object

```
SearchResults searchForWord
 (String word) {
 return new SearchResults
 (..., word, ..., StreamSupport
 .stream(new WordMatchSpliterator
 (mInput, word),
 false)
 .collect(toList()));
}
```

*This stream is collected into a list  
of SearchResults.Result objects*



# Applying Java Spliterator in SimpleSearchStream

---

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
private final Matcher mWordMatcher;

public WordMatchSpliterator(String input, String word) {
 ...
 String regexWord = "\\b" + word.trim() + "\\b";
 mWordMatcher =
 Pattern.compile(regexWord,
 Pattern.CASE_INSENSITIVE)
 .matcher(input); ...
```

---

See [SimpleSearchStream/src/main/java/search/WordMatchSpliterator.java](#)

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
private final Matcher mWordMatcher;
```

*The extending class need only implement tryAdvance()*

```
public WordMatchSpliterator(String input, String word) {
 ...
 String regexWord = "\\b" + word.trim() + "\\b";
 mWordMatcher =
 Pattern.compile(regexWord,
 Pattern.CASE_INSENSITIVE)
 .matcher(input); ...
```

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
private final Matcher mWordMatcher;
```

*An engine that performs regex match operations on a character sequence.*

```
public WordMatchSpliterator(String input, String word) {
 ...
 String regexWord = "\\b" + word.trim() + "\\b";

 mWordMatcher =
 Pattern.compile(regexWord,
 Pattern.CASE_INSENSITIVE)
 .matcher(input); ...
```

See [docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html](https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html)

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
private final Matcher mWordMatcher;
```

*Constructor is passed the input string & a given word to search for matches.*

```
public WordMatchSpliterator(String input, String word) {
 ...
 String regexWord = "\\b" + word.trim() + "\\b";

 mWordMatcher =
 Pattern.compile(regexWord,
 Pattern.CASE_INSENSITIVE)
 .matcher(input); ...
```

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
private final Matcher mWordMatcher;

public WordMatchSpliterator(String input, String word) {
 ...
 String regexWord = "\\b" + word.trim() + "\\b";
 mWordMatcher =
 Pattern.compile(regexWord,
 Pattern.CASE_INSENSITIVE)
 .matcher(input); ...
```

*This regex only matches  
a "word", not a substring*

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
private final Matcher mWordMatcher;

public WordMatchSpliterator(String input, String word) {
 ...
 String regexWord = "\\b" + word.trim() + "\\b";
 mWordMatcher =
 Pattern.compile(regexWord,
 Pattern.CASE_INSENSITIVE)
 .matcher(input); ...
}
```

*Compile the regex & create a matcher for the input string*

See [docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html](https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html)

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
 ...
 public boolean tryAdvance(Consumer<? super Result> action) {
 if (!mWordMatcher.find())
 return false;
 else {
 action.accept(new Result(mWordMatcher.start()));
 return true;
 }
 }
}
```

*Called by the Java streams framework to attempt to advance the spliterator by one word match*

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

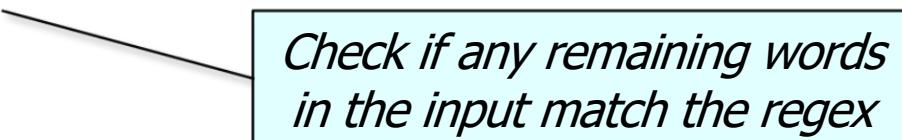
```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
 ...
 public boolean tryAdvance(Consumer<? super Result> action) {
 if (!mWordMatcher.find())
 return false;
 else {
 action.accept(new Result(mWordMatcher.start()));
 return true;
 }
 }
}
```

*Passes the result (if any) back "by reference" to the streams framework*

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
 ...
 public boolean tryAdvance(Consumer<? super Result> action) {
 if (!mWordMatcher.find())
 return false;
 else {
 action.accept(new Result(mWordMatcher.start()));
 return true;
 }
 }
}
```



*Check if any remaining words in the input match the regex*

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
 ...
 public boolean tryAdvance(Consumer<? super Result> action) {
 if (!mWordMatcher.find())
 return false;
 else {
 action.accept(new Result(mWordMatcher.start()));
 return true;
 }
 }
}
```

*Inform the streams framework to cease calling tryAdvance() if there's no match*

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
 ...
 public boolean tryAdvance(Consumer<? super Result> action) {
 if (!mWordMatcher.find())
 return false;
 else {
 action.accept(new Result(mWordMatcher.start()));
 return true;
 }
 }
}
```



*accept() stores the index in the input string where the match occurred, which is returned to the streams framework*

# Applying Java Spliterator in SimpleSearchStream

- WordMatchSpliterator uses Java regex to create a stream of SearchResults. Result objects that match the # of times a word appears in an input string

```
class WordMatchSpliterator
 extends Spliterators.AbstractSpliterator<Result> {
 ...
 public boolean tryAdvance(Consumer<? super Result> action) {
 if (!mWordMatcher.find())
 return false;

 else {
 action.accept(new Result(mWordMatcher.start()));
 return true;
 }
 }
}
```

*Inform the streams framework  
to continue calling tryAdvance()*

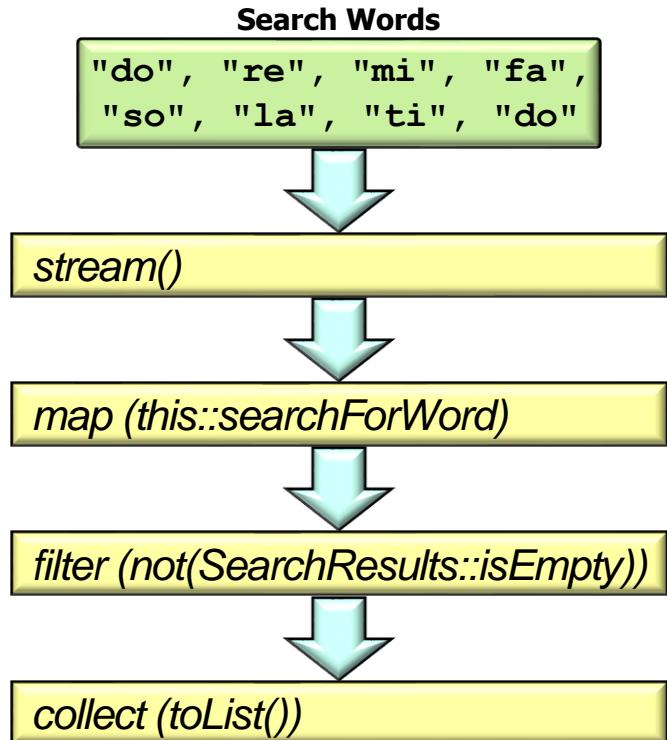
# Applying Java Spliterator in SimpleSearchStream

- Here's a recap of how searchForWord() uses WordMatchSpliterator

```
SearchResults searchForWord
 (String word) {

 return new SearchResults
 (..., word, ..., StreamSupport
 .stream(new WordMatchSpliterator
 (mInput, word),
 false)
 .collect(toList()));
}
```

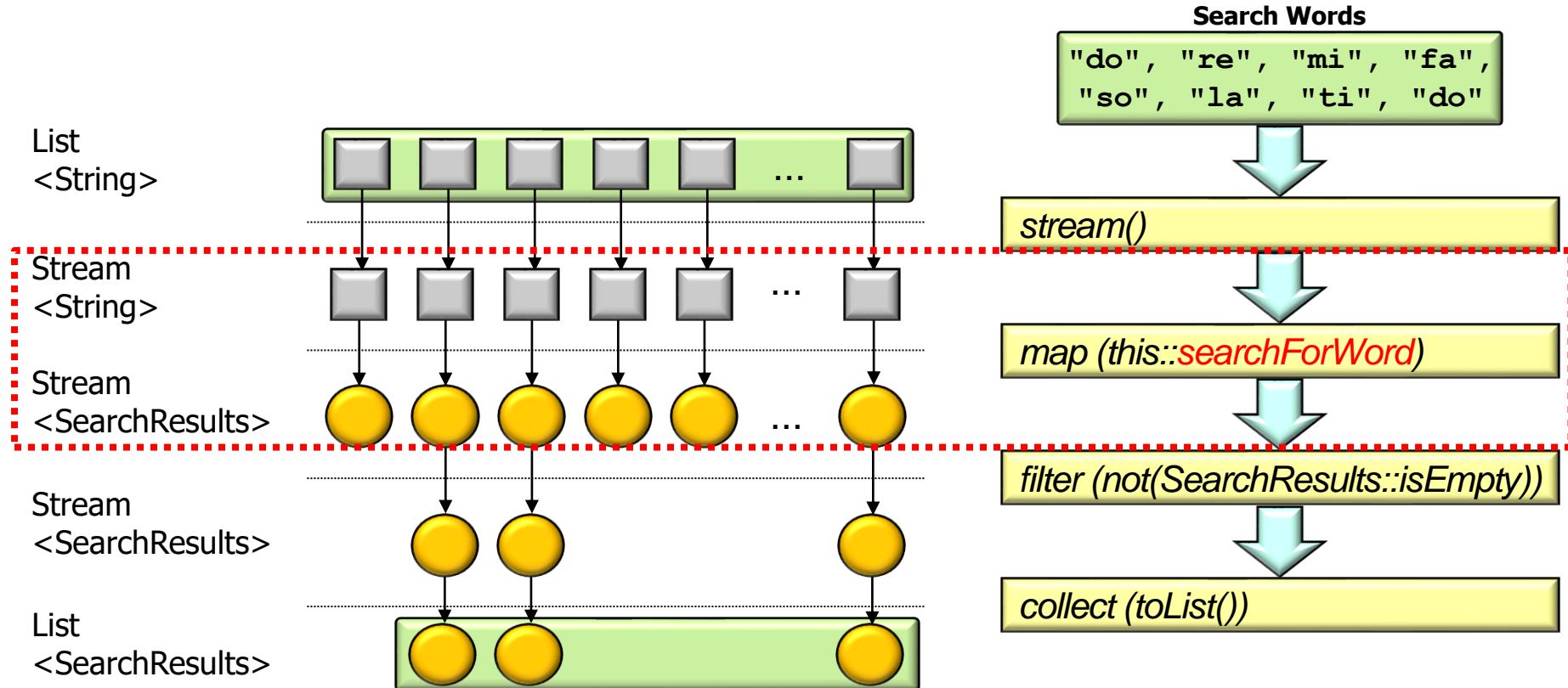
*StreamSupport.stream() creates a sequential  
stream via the WordMatchSpliterator class*



See [SimpleSearchStream/src/main/java/search/WordMatchSpliterator.java](#)

# Applying Java Spliterator in SimpleSearchStream

- Here's the output that searchForWord() & WordMatchSpliterator produce



---

# End of Apply Java Streams Spliterators

# **Understand Java Streams**

# **Non-Concurrent Collectors**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of non-concurrent collectors for sequential streams

## Interface Collector<T,A,R>

### Type Parameters:

T - the type of input elements to the reduction operation

A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)

R - the result type of the reduction operation

public interface **Collector<T,A,R>**

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a Collection; concatenating strings using a StringBuilder; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class **Collectors** provides implementations of many common mutable reductions.

A Collector is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html)

---

# Overview of Non-Concurrent Collectors

# Overview of Non-Concurrent Collectors

- The collect() terminal operation uses a collector to accumulate stream elements into mutable result containers.

```
void runCollectToList() {
 List<String> characters = Arrays
 .asList("horatio", "laertes",
 "Hamlet, ...);

 List<String> results =
 characters
 .stream()
 .filter(s ->
 toLowerCase(... == 'h')
 .map(this::capitalize)
 .sorted()
 .collect(toList()); ...
```

*Collect the results into a ArrayList*

# Overview of Non-Concurrent Collectors

- The collect() terminal operation uses a collector to accumulate stream elements into mutable result containers.
  - Collector is defined by a generic interface



## Interface Collector<T,A,R>

### Type Parameters:

T - the type of input elements to the reduction operation

A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)

R - the result type of the reduction operation

### public interface Collector<T,A,R>

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a Collection; concatenating strings using a StringBuilder; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class Collectors provides implementations of many common mutable reductions.

A Collector is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

# Overview of Non-Concurrent Collectors

- Collector implementations can either be non-concurrent or concurrent based on their characteristics

## Enum Collector.Characteristics

java.lang.Object  
java.lang.Enum<Collector.Characteristics>  
java.util.stream.Collector.Characteristics

### All Implemented Interfaces:

Serializable, Comparable<Collector.Characteristics>

### Enclosing Interface:

Collector<T,A,R>

---

public static enum Collector.Characteristics  
extends Enum<Collector.Characteristics>

Characteristics indicating properties of a Collector, which can be used to optimize reduction implementations.

## Enum Constant Summary

### Enum Constants

#### Enum Constant and Description

##### CONCURRENT

Indicates that this collector is *concurrent*, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.

##### IDENTITY\_FINISH

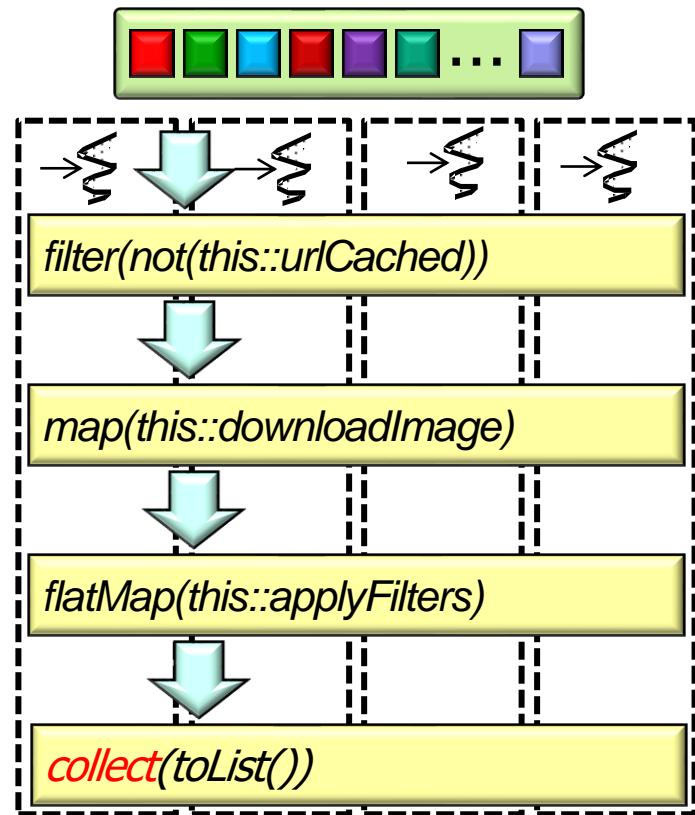
Indicates that the finisher function is the identity function and can be elided.

##### UNORDERED

Indicates that the collection operation does not commit to preserving the encounter order of input elements.

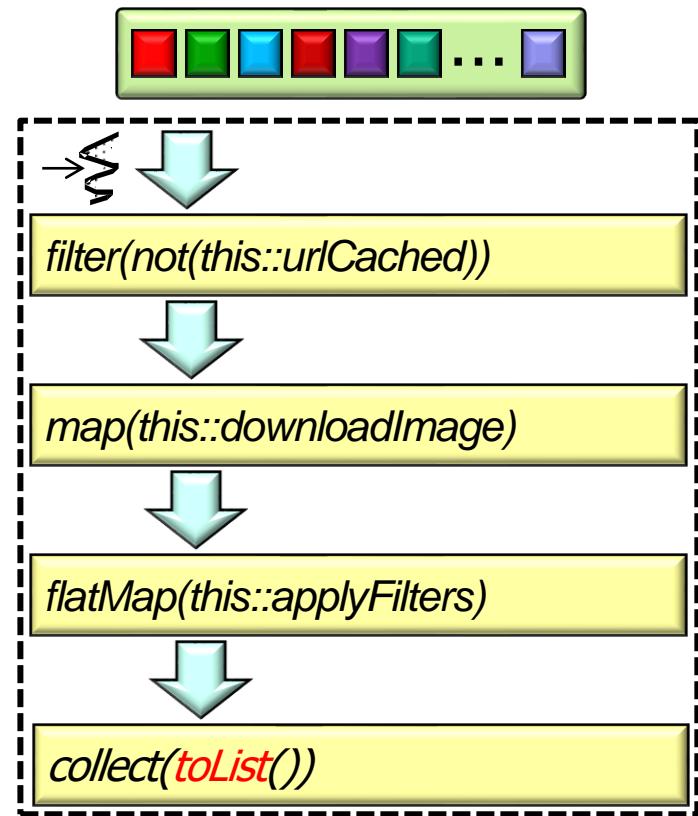
# Overview of Non-Concurrent Collectors

- Collector implementations can either be non-concurrent or concurrent based on their characteristics
  - This distinction is only relevant for *parallel* streams



# Overview of Non-Concurrent Collectors

- Collector implementations can either be non-concurrent or concurrent based on their characteristics
  - This distinction is only relevant for *parallel* streams
  - Our focus here is on non-concurrent collectors for sequential streams



Non-concurrent & concurrent collectors for parallel streams are covered later

# Overview of Non-Concurrent Collectors

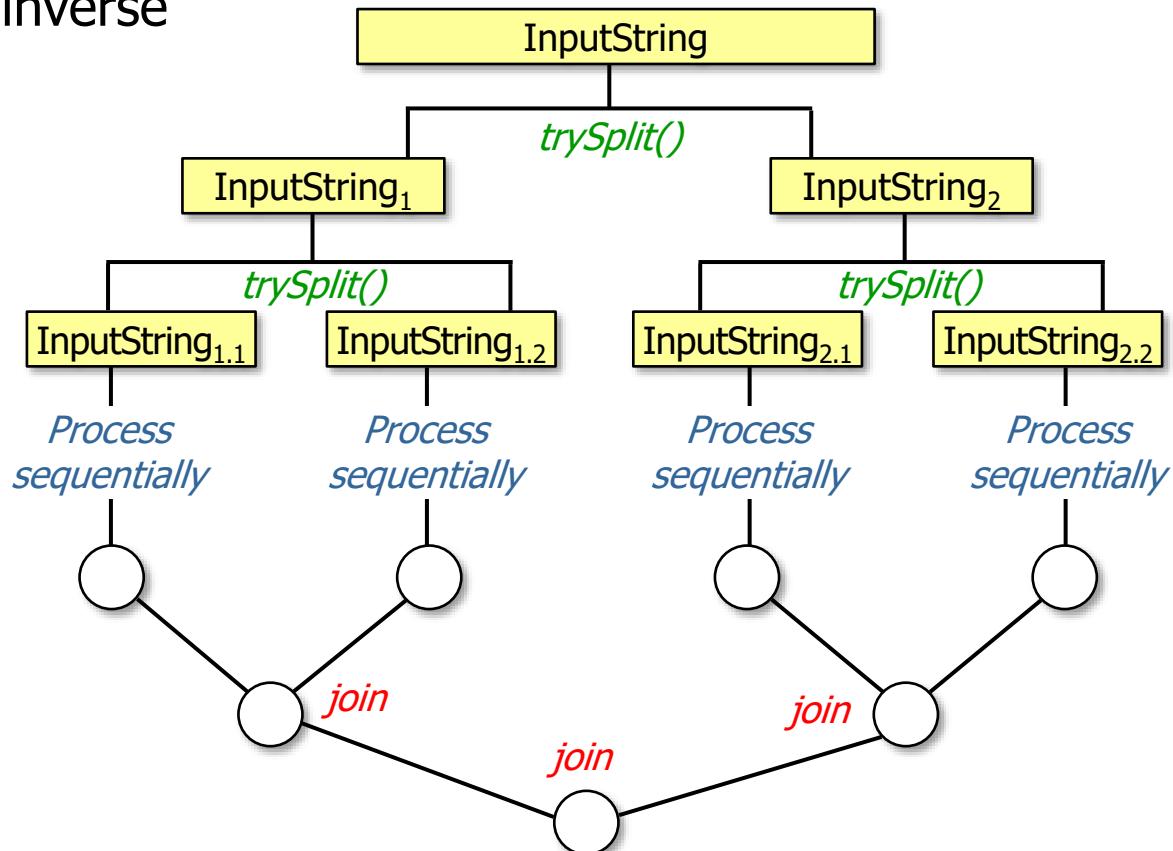
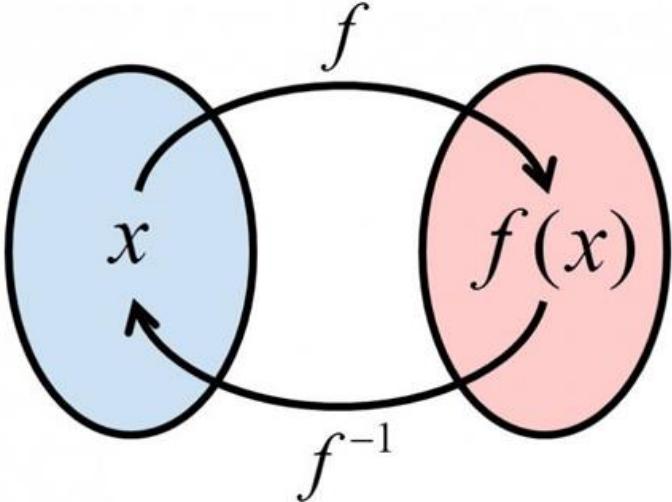
---

- A non-concurrent collector for a sequential stream simply accumulates elements into a mutable result container



# Overview of Non-Concurrent Collectors

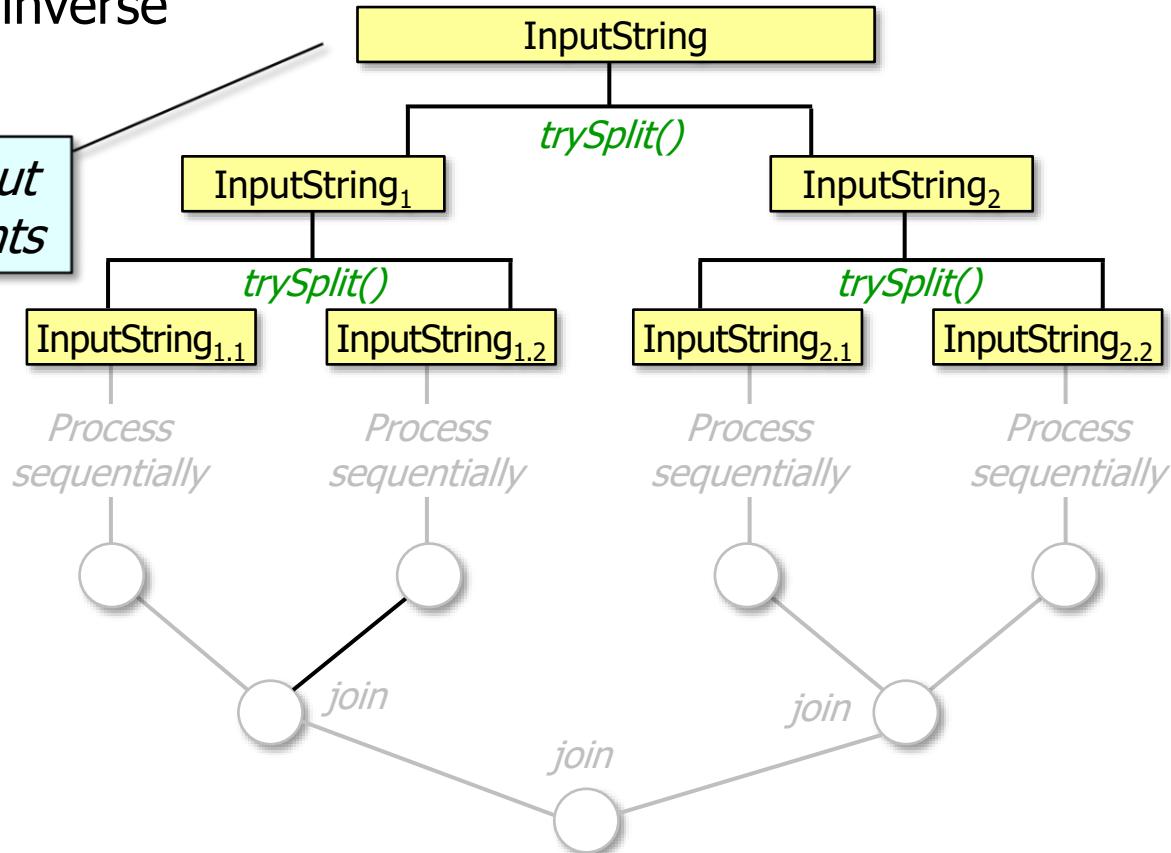
- A collector is essentially the inverse of a spliterator



# Overview of Non-Concurrent Collectors

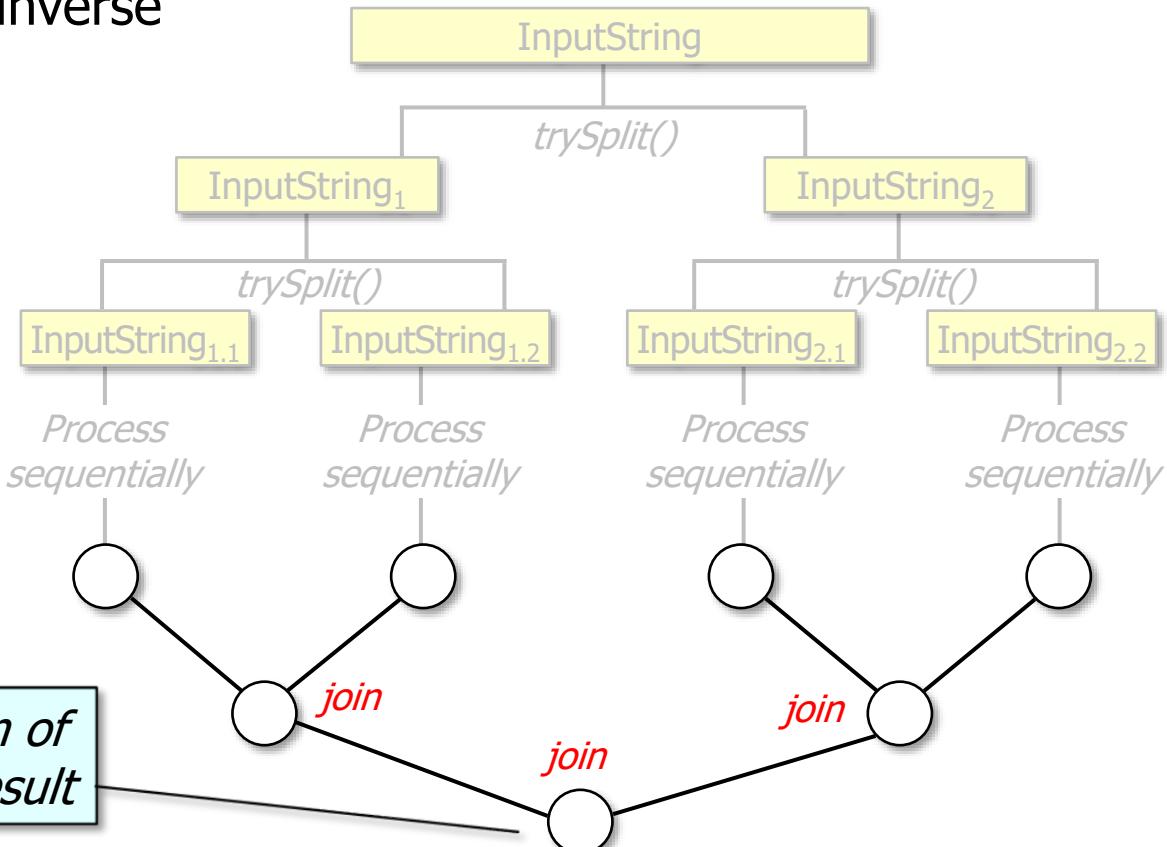
- A collector is essentially the inverse of a spliterator

*A spliterator partitions one input source into a stream of elements*



# Overview of Non-Concurrent Collectors

- A collector is essentially the inverse of a splitterator



---

End of Understand  
Java Streams Non-  
Concurrent Collectors

# **Understand the Java Streams Non-Concurrent Collector API**

**Douglas C. Schmidt**

**d.schmidt@vanderbilt.edu**

**www.dre.vanderbilt.edu/~schmidt**

**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

---

- Understand the structure & functionality of non-concurrent collectors for sequential streams
- Know the API for non-concurrent collectors

<<Java Interface>>

**Collector<T,A,R>**

---

- `supplier():Supplier<A>`
- `accumulator():BiConsumer<A,T>`
- `combiner():BinaryOperator<A>`
- `finisher():Function<A,R>`
- `characteristics():Set<Characteristics>`

---

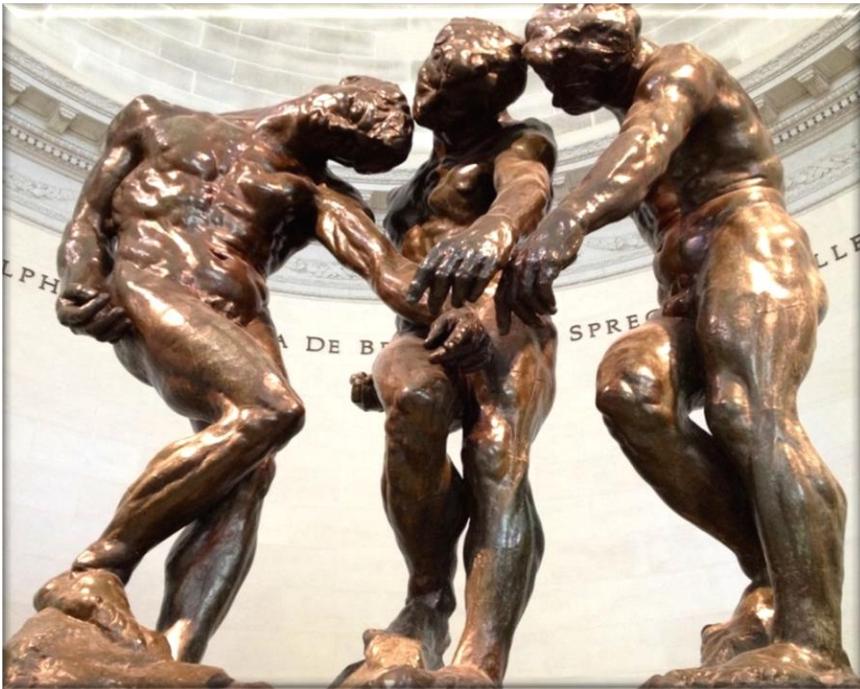
The same API is also used for concurrent collectors!

---

# The Non-Concurrent Collector API

# The Non-Concurrent Collector API

- The Collector interface defines three generic types



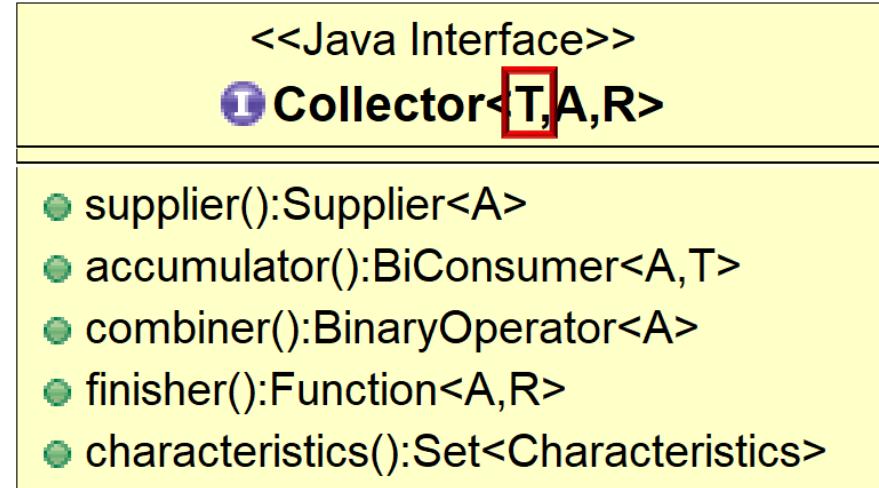
<<Java Interface>>

**Collector<T,A,R>**

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

# The Non-Concurrent Collector API

- The Collector interface defines three generic types
  - **T** - The type of elements available in the stream
    - e.g., Long, String, SearchResults, etc.



# The Non-Concurrent Collector API

- The Collector interface defines three generic types
  - **T**
  - **A** – The type of mutable accumulator object to use for collecting elements
    - e.g., List of T (implemented via ArrayList, LinkedList, etc.)

<<Java Interface>>

 **Collector<T,A,R>**

---

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

# The Non-Concurrent Collector API

- The Collector interface defines three generic types
  - T
  - A
  - R – The type of the final result
    - e.g., List of T

<<Java Interface>>

 **Collector<T,A,R>**

---

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

The type of R & the type of A may or may not be different!

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface



<<Java Interface>>

 **Collector<T,A,R>**

- `supplier():Supplier<A>`
- `accumulator():BiConsumer<A,T>`
- `combiner():BinaryOperator<A>`
- `finisher():Function<A,R>`
- `characteristics():Set<Characteristics>`

Again, this discussion assumes we're implementing a *non-concurrent* collector

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()** – provides a stream with additional information used for internal optimizations

<<Java Interface>>

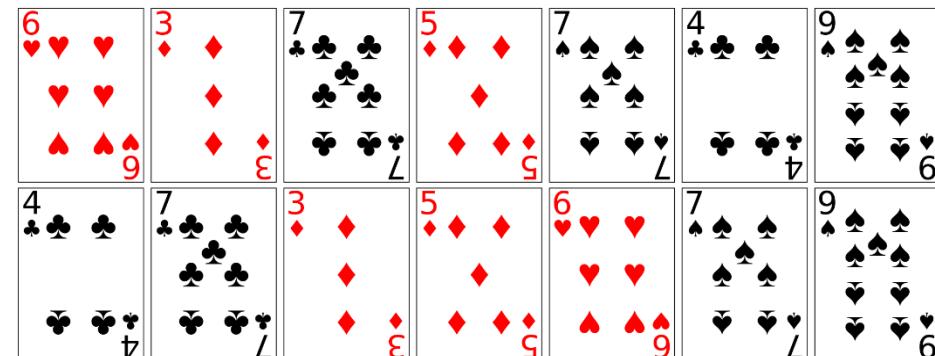
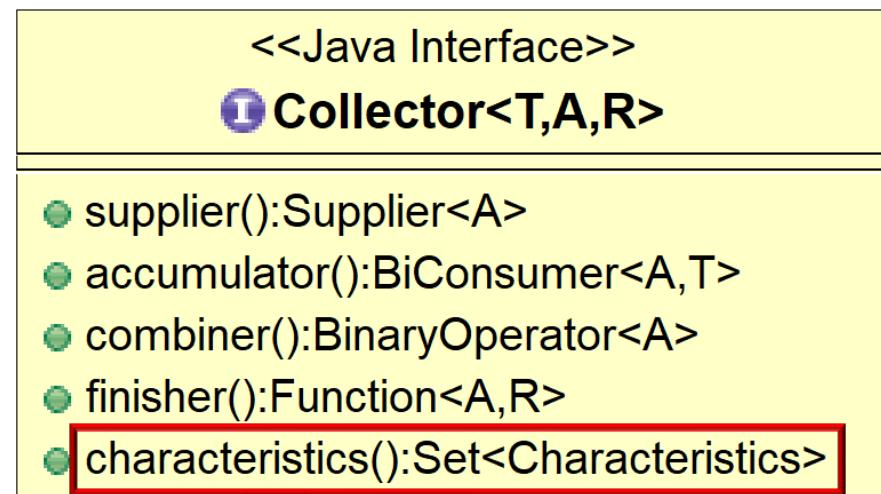
**Collector<T,A,R>**

---

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()** – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
      - The collector need not preserve the encounter order



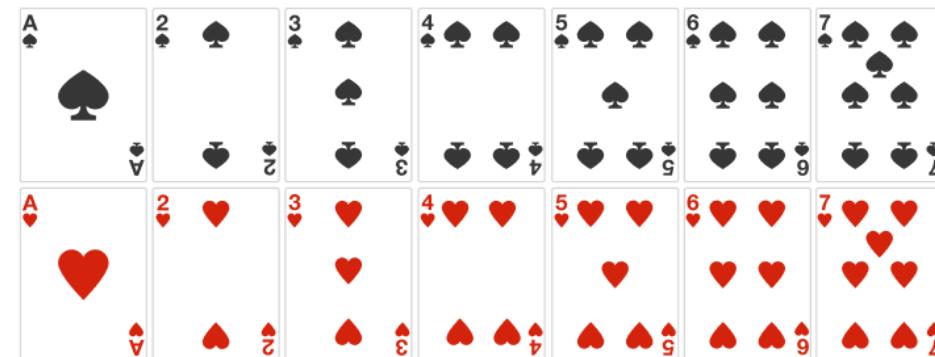
# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()** – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
      - The collector need not preserve the encounter order

<<Java Interface>>

**Collector<T,A,R>**

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



A collector may preserve encounter order if it incurs no additional overhead

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()** – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTITY\_FINISH
      - The finisher() is the identity function so it can be a no-op
        - e.g., finisher() just returns null

| <<Java Interface>>                                                                                                |  |
|-------------------------------------------------------------------------------------------------------------------|--|
|  <b>Collector&lt;T,A,R&gt;</b> |  |
| • supplier():Supplier<A>                                                                                          |  |
| • accumulator():BiConsumer<A,T>                                                                                   |  |
| • combiner():BinaryOperator<A>                                                                                    |  |
| • finisher():Function<A,R>                                                                                        |  |
| • characteristics():Set<Characteristics>                                                                          |  |



# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()** – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTITY\_FINISH
    - CONCURRENT
      - The accumulator method is called concurrently on the result container

<<Java Interface>>

**I Collector<T,A,R>**

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

*The mutable result container must be synchronized!!*



# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()** – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTITY\_FINISH
    - CONCURRENT
      - The accumulator method is called concurrently on the result container

<<Java Interface>>

**I Collector<T,A,R>**

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



We're focusing on a non-concurrent collector, which doesn't enable CONCURRENT

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()** – provides a stream with additional information used for internal optimizations, e.g.

*Any/all characteristics can be set using EnumSet.of()*

```
Set characteristics () {
 return Collections.unmodifiableSet
 (EnumSet.of(Collector.Characteristics.CONCURRENT,
 Collector.Characteristics.UNORDERED,
 Collector.Characteristics.IDENTITY_FINISH)) ;
}
```

<<Java Interface>>

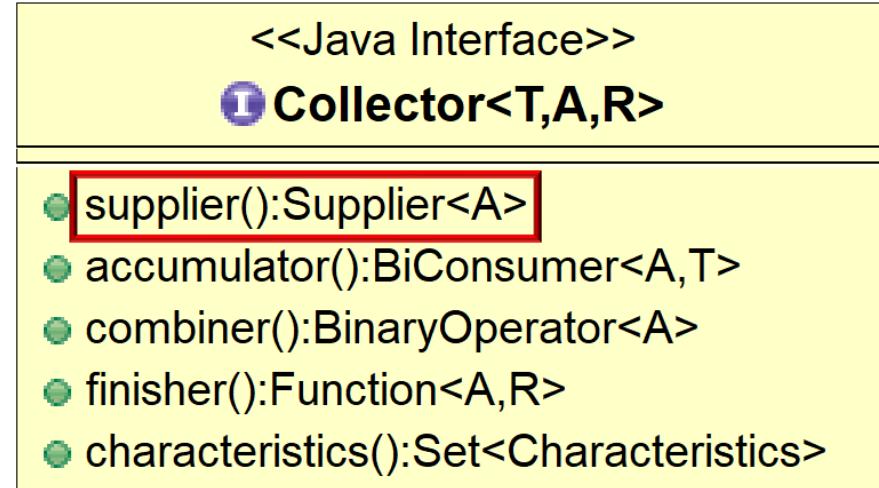
 **Collector<T,A,R>**

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

See [docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html](https://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html)

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()**
  - **supplier()** – returns a supplier that acts as a factory to generate an empty result container



# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()**
  - **supplier()** – returns a supplier that acts as a factory to generate an empty result container, e.g.

```
Supplier<List> supplier() {
 return ArrayList::new;
}
```

<<Java Interface>>

**I Collector<T,A,R>**

- **supplier():Supplier<A>**
- **accumulator():BiConsumer<A,T>**
- **combiner():BinaryOperator<A>**
- **finisher():Function<A,R>**
- **characteristics():Set<Characteristics>**

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()**
  - **supplier()**
  - **accumulator()** – returns a bi-consumer that adds a new element to an existing result container, e.g.

```
BiConsumer<List, Integer> accumulator() {
 return List::add;
}
```

<<Java Interface>>

**I Collector<T,A,R>**

- supplier():Supplier<A>
- **accumulator():BiConsumer<A,T>**
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

*A non-concurrent collector needs no synchronization*



See [docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#add](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#add)

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()**
  - **supplier()**
  - **accumulator()**
  - **combiner()** – returns a binary operator that merges two result containers together, e.g.

```
BinaryOperator<List> combiner() {
 return (one, another) -> {
 one.addAll(another);
 return one;
 };
}
```

<<Java Interface>>

**I Collector<T,A,R>**

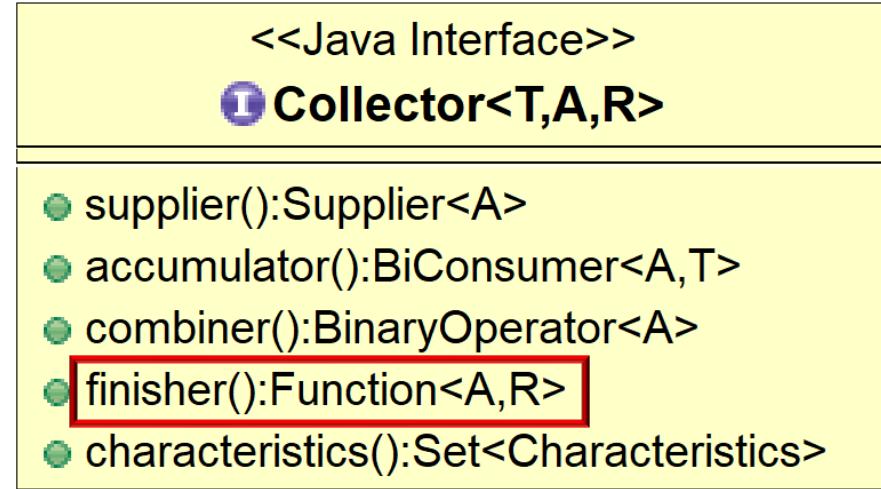
---

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- **combiner():BinaryOperator<A>**
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

This combiner() will not be called for a sequential stream..

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()**
  - **supplier()**
  - **accumulator()**
  - **combiner()**
  - **finisher()** – returns a function that converts the result container to final result type, e.g.
    - `return Function.identity()`



# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - **characteristics()**
  - **supplier()**
  - **accumulator()**
  - **combiner()**
  - **finisher()** – returns a function that converts the result container to final result type, e.g.
    - `return Function.identity()`
    - `return null;`

<<Java Interface>>

**I Collector<T,A,R>**

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>



*Should be a no-op if IDENTITY\_FINISH characteristic is set*

# The Non-Concurrent Collector API

- Five factory methods are defined in the Collector interface
  - `characteristics()`
  - `supplier()`
  - `accumulator()`
  - `combiner()`
  - **finisher()** – returns a function that converts the result container to final result type, e.g.
    - `return Function.identity()`
    - `return null;`

```
Stream<BigFraction>
 .generate(() ->
 makeBigFraction
 (new Random(), false))
 .limit(sMAX_FRACTIONS)

 .map(reduceAndMultiplyFraction)
 .collect(FuturesCollector<BigFraction>
 .toFuture()))

 .thenAccept
 (this::sortAndPrintList);
```

*finisher() can also be much more interesting!*

See [Java8/ex19/src/main/java/utils/FuturesCollector.java](https://github.com/Java8/ex19/blob/main/src/main/java/utils/FuturesCollector.java)

---

# End of Understand the Java Streams Non-Concurrent Collector API

# Learn How Pre-defined Non-Concurrent Collectors are Implemented

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

---

- Understand the structure & functionality of non-concurrent collectors for sequential streams
- Know the API for non-concurrent collectors
- Recognize how pre-defined non-concurrent collectors are implemented in the JDK

## Class Collectors

java.lang.Object  
java.util.stream.Collectors

```
public final class Collectors
extends Object
```

Implementations of `Collector` that implement various useful reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

The following are examples of using the predefined collectors to perform common mutable reduction tasks:

---

# How Pre-defined Non-Concurrent Collectors are Implemented

# How Pre-defined Non-Concurrent Collectors are Implemented

- Collectors is a utility class whose factory methods create collectors for common collection types

## Class Collectors

`java.lang.Object`  
`java.util.stream.Collectors`

```
public final class Collectors
extends Object
```

Implementations of `Collector` that implement various useful reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

The following are examples of using the predefined collectors to perform common mutable reduction tasks:

# How Pre-defined Non-Concurrent Collectors are Implemented

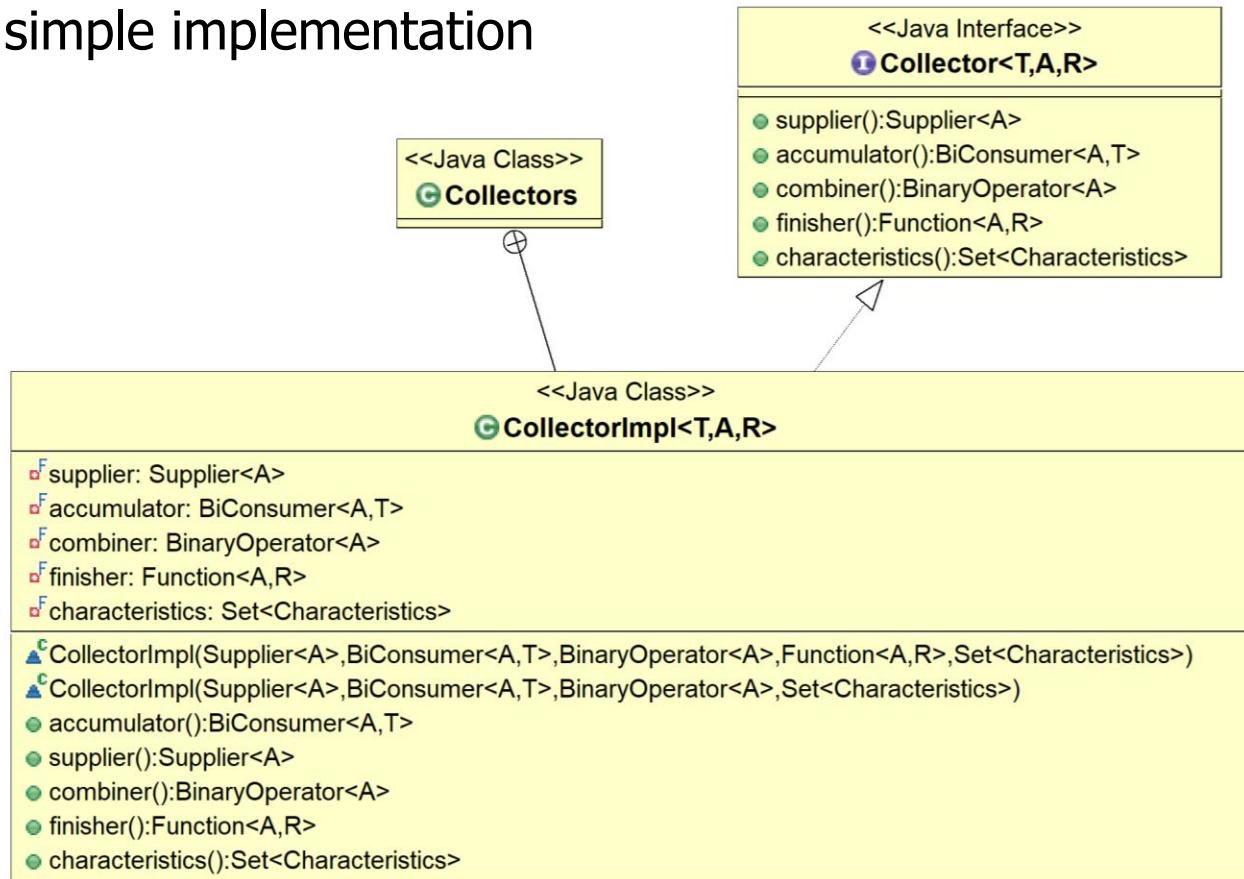
- Collectors is a utility class whose factory methods create collectors for common collection types
  - A utility class is final, has only static methods, no (non-static) state, & a private constructor

```
<<Java Class>>
Collectors

Collectors()
toCollection(Supplier<C>):Collector<T,?>,C>
toList():Collector<T,?,List<T>>
toSet():Collector<T,?,Set<T>>
joining():Collector<CharSequence,?,String>
joining(CharSequence):Collector<CharSequence,?,String>
joining(CharSequence,CharSequence,CharSequence):Collector<CharSequence,?,String>
mapping(Function<? super T,? extends U>,Collector<? super U,A,R>):Collector<T,?,R>
collectingAndThen(Collector<T,A,R>,Function<R,RR>):Collector<T,A,RR>
counting():Collector<T,?,Long>
minBy(Comparator<? super T>):Collector<T,?,Optional<T>>
maxBy(Comparator<? super T>):Collector<T,?,Optional<T>>
summingInt(TolIntFunction<? super T>):Collector<T,?,Integer>
summingLong(ToLongFunction<? super T>):Collector<T,?,Long>
summingDouble(ToDoubleFunction<? super T>):Collector<T,?,Double>
averagingInt(TolIntFunction<? super T>):Collector<T,?,Double>
averagingLong(ToLongFunction<? super T>):Collector<T,?,Double>
averagingDouble(ToDoubleFunction<? super T>):Collector<T,?,Double>
reducing(T,BinaryOperator<T>):Collector<T,?,T>
reducing(BinaryOperator<T>):Collector<T,?,Optional<T>>
reducing(U,Function<? super T,? extends U>,BinaryOperator<U>):Collector<T,?,U>
groupingBy(Function<? super T,? extends K>):Collector<T,?,Map<K,List<T>>>
toMap(Function<? super T,? extends K>,Function<? super T,? extends U>):Collector<T,?,Map<K,U>>
summarizingInt(TolIntFunction<? super T>):Collector<T,?,IntSummaryStatistics>
summarizingLong(ToLongFunction<? super T>):Collector<T,?,LongSummaryStatistics>
summarizingDouble(ToDoubleFunction<? super T>):Collector<T,?,DoubleSummaryStatistics>
```

# How Pre-defined Non-Concurrent Collectors are Implemented

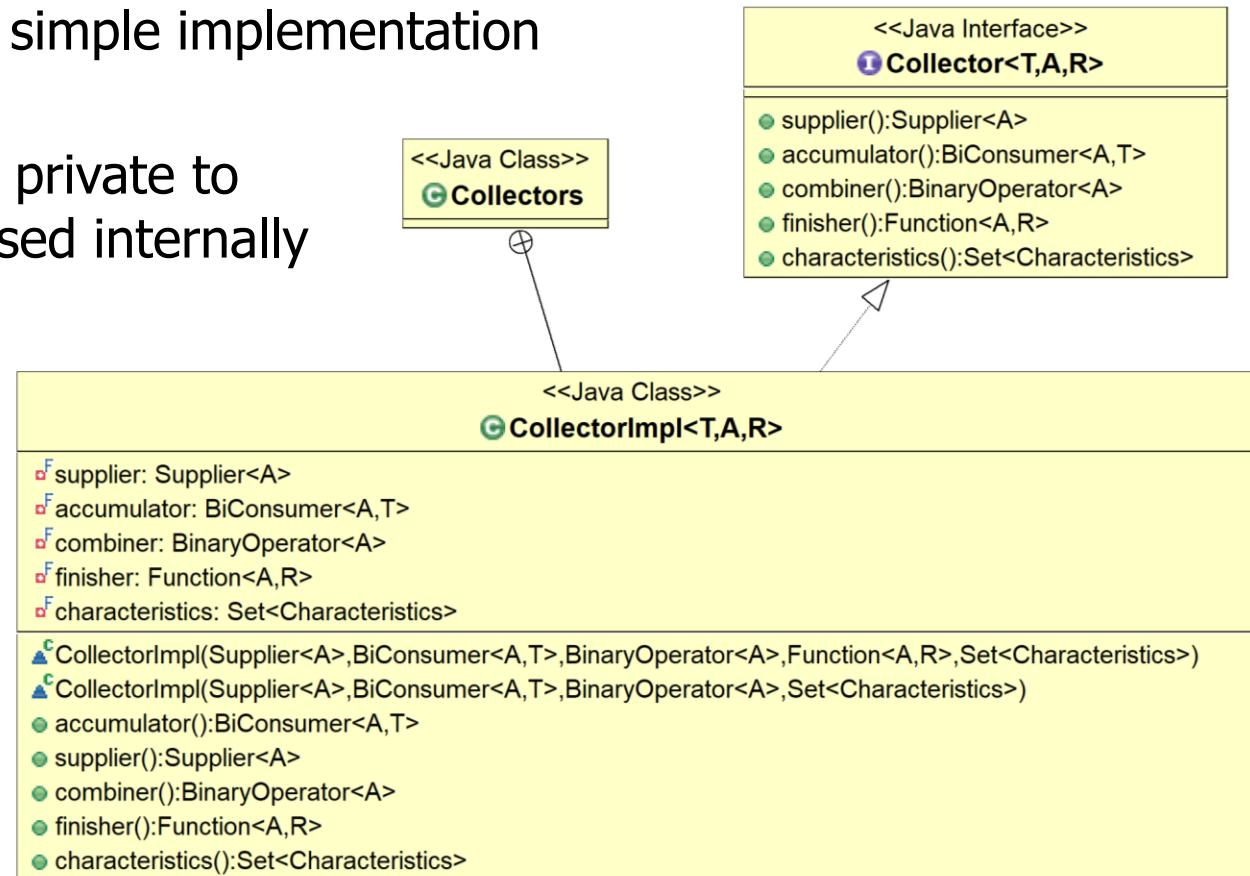
- CollectorImpl defines a simple implementation class for a Collector



See [openjdk/8-b132/java/util/stream/Collectors.java#Collectors.CollectorImpl](https://openjdk/8-b132/java/util/stream/Collectors.java#Collectors.CollectorImpl)

# How Pre-defined Non-Concurrent Collectors are Implemented

- CollectorImpl defines a simple implementation class for a Collector
  - However, this class is private to Collectors & is only used internally



PRIVATE

# How Pre-defined Non-Concurrent Collectors are Implemented

- `Collectors.toList()` uses `CollectorImpl` to return a non-concurrent collector that accumulates input elements into a new (ArrayList)

```
final class Collectors {
 ...
 public static <T> Collector
 <T, ?, List<T>>
 toList() {
 return new CollectorImpl<>
 ((Supplier<List<T>>)
 ArrayList::new,
 List::add,
 (left, right) -> {
 left.addAll(right);
 return left;
 },
 CH_ID);
 } ...
```

# How Pre-defined Non-Concurrent Collectors are Implemented

- `Collectors.toList()` uses `CollectorImpl` to return a non-concurrent collector that accumulates input elements into a new (Array)List

*The supplier constructor reference*

```
final class Collectors {
 ...
 public static <T> Collector
 <T, ?, List<T>>
 toList() {
 return new CollectorImpl<>
 ((Supplier<List<T>>)
 ArrayList::new,
 List::add,
 (left, right) -> {
 left.addAll(right);
 return left;
 },
 CH_ID);
 } ...
}
```

# How Pre-defined Non-Concurrent Collectors are Implemented

- `Collectors.toList()` uses `CollectorImpl` to return a non-concurrent collector that accumulates input elements into a new (Array)List

*The accumulator method reference*

```
final class Collectors {
 ...
 public static <T> Collector
 <T, ?, List<T>>
 toList() {
 return new CollectorImpl<>
 ((Supplier<List<T>>)
 ArrayList::new,
 List::add,
 (left, right) -> {
 left.addAll(right);
 return left;
 },
 CH_ID);
 } ...
}
```

# How Pre-defined Non-Concurrent Collectors are Implemented

- `Collectors.toList()` uses `CollectorImpl` to return a non-concurrent collector that accumulates input elements into a new (Array)List

```
final class Collectors {
 ...
 public static <T> Collector
 <T, ?, List<T>>
 toList() {
 return new CollectorImpl<>
 ((Supplier<List<T>>)
 ArrayList::new,
 List::add,
 (left, right) -> {
 left.addAll(right);
 return left;
 },
 CH_ID);
 } ...
}
```

*The combiner lambda expression*

This combiner is only used for parallel streams

# How Pre-defined Non-Concurrent Collectors are Implemented

- `Collectors.toList()` uses `CollectorImpl` to return a non-concurrent collector that accumulates input elements into a new (Array)List

```
final class Collectors {
 ...
 public static <T> Collector
 <T, ?, List<T>>
 toList() {
 return new CollectorImpl<>
 ((Supplier<List<T>>)
 ArrayList::new,
 List::add,
 (left, right) -> {
 left.addAll(right);
 return left;
 },
 CH_ID);
 } ...
}
```

*Characteristics set*

`CH_ID` is defined as `Collector.Characteristics.IDENTITY_FINISH`

# How Pre-defined Non-Concurrent Collectors are Implemented

- Collector.of() defines a simple public factory method that implements a Collector

```
interface Collector<T, A, R> { ...
 static<T, R> Collector<T, R, R> of
 (Supplier<R> supplier,
 BiConsumer<R, T> accumulator,
 BinaryOperator<R> combiner,
 Characteristics... chars) {
 ...
 return new Collectors
 .CollectorImpl<>
 (supplier,
 accumulator,
 combiner,
 chars);
 } ...
```

*This of() method is passed four params (last param is optional)*

# How Pre-defined Non-Concurrent Collectors are Implemented

- Collector.of() defines a simple public factory method that implements a Collector

```
interface Collector<T, A, R> { ...
 static<T, R> Collector<T, R, R> of
 (Supplier<R> supplier,
 BiConsumer<R, T> accumulator,
 BinaryOperator<R> combiner,
 Function<A,R> finisher,
 Characteristics... chars) {
 ...
 return new Collectors
 .CollectorImpl<>
 (supplier,
 accumulator,
 combiner,
 finisher,
 chars); ...
}
```

*This of() method is passed five params (last param is optional)*

# How Pre-defined Non-Concurrent Collectors are Implemented

- Collector.of() defines a simple public factory method that implements a Collector
  - Both of() versions internally use the private CollectorImpl class

```
interface Collector<T, A, R> { ...
 static<T, R> Collector<T, R, R> of
 (Supplier<R> supplier,
 BiConsumer<R, T> accumulator,
 BinaryOperator<R> combiner,
 Function<A,R> finisher,
 Characteristics... chars) {
 ...
 return new Collectors
 .CollectorImpl<>
 (supplier,
 accumulator,
 combiner,
 finisher,
 chars); ...
 }
}
```

---

# End of Learn How Pre-defined Non-Concurrent Collectors are Implemented

# Learn How to Implement Custom Non-Concurrent Collectors

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of non-concurrent collectors for sequential streams
- Know the API for non-concurrent collectors
- Recognize how to apply pre-defined non-concurrent collectors
- Be able to implement custom non-concurrent collectors

```
interface Collector<T, A, R> {
 ...
 static<T, R>
 Collector<T, R, R> of(
 Supplier<R> supplier,
 BiConsumer<R, T>
 accumulator,
 BinaryOperator<R>
 combiner,
 Function<A,R>
 finisher,
 Characteristics...
 chars) {
 ...
}
```

# Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of non-concurrent collectors for sequential streams
- Know the API for non-concurrent collectors
- Recognize how to apply pre-defined non-concurrent collectors
- Be able to implement custom non-concurrent collectors
  - e.g., we analyze several implementations of non-concurrent collectors from the SimpleSearchStream program



---

# Implementing Custom Non-Concurrent Collectors (Part 1)

# Implementing Custom Non-Concurrent Collectors (Part 1)

- Collector.of() can implement custom collectors that have pithy lambdas

```
public String toString() {
 ...
 mList.stream()
 .collect(Collector.of(() -> new StringJoiner(" | ") ,
 (j, r) -> j.add(r.toString())) ,
 StringJoiner::merge,
 StringJoiner::toString));
 ...
}
```



```
(j, r) -> j.add(r.toString()) ,
 StringJoiner::merge,
 StringJoiner::toString));
 ...
}
}
```

# Implementing Custom Non-Concurrent Collectors (Part 1)

- The SearchResults.toString() method uses Collector.of() to format results

```
public String toString() {
 ...
 mList.stream()
 .collect(Collector.of(() -> new StringJoiner(" | ") ,
 (j, r) -> j.add(r.toString()) ,
 StringJoiner::merge,
 StringJoiner::toString));
 ...
}
```



(j, r) -> j.add(r.toString()) ,

*SearchResults's custom  
collector formats itself*

StringJoiner::merge,  
StringJoiner::toString));  
...

See [SimpleSearchStream/src/main/java/search/SearchResults.java](#)

# Implementing Custom Non-Concurrent Collectors (Part 1)

- The SearchResults.toString() method uses Collector.of() to format results

```
public String toString() {
 ...
 mList.stream()
 .collect(Collector.of(() -> new StringJoiner(" | ") ,

```

*Factory method creates a new collector  
via the five-param of() method version*

```
(j, r) -> j.add(r.toString()) ,
```

```
StringJoiner::merge,
StringJoiner::toString)); ...
```



# Implementing Custom Non-Concurrent Collectors (Part 1)

- The SearchResults.toString() method uses Collector.of() to format results

```
public String toString() {
 ...
 mList.stream()
 .collect(Collector.of(() -> new StringJoiner(" | ") ,

```

*This lambda supplier creates  
the mutable result container*

```
(j, r) -> j.add(r.toString()) ,
 }
 .StringJoiner::merge,
 .StringJoiner::toString)); ...
}
```

979|1025|1219|1259|  
1278|1300|1351|1370|1835|  
1875|1899|1939|2266|2295]  
Word "Ti" matched at index [237|994|1272|1294|1364|1850|  
1860|1912|1915|1952|1955|  
2299]  
Word "La" matched at index [234|417|658|886|991|1207|  
1247|1269|1291|1339|1361|  
1742|1847|1863|1909|1949|  
2161|2254|2276|2283]...  
Ending SimpleSearchStream"/>

The phone's status bar shows the time as 8:50 AM.

See [docs.oracle.com/javase/8/docs/api/java/util/StringJoiner.html](https://docs.oracle.com/javase/8/docs/api/java/util/StringJoiner.html)

# Implementing Custom Non-Concurrent Collectors (Part 1)

- The SearchResults.toString() method uses Collector.of() to format results

```
public String toString() {
 ...
 mList.stream()
 .collect(Collectors.of(() -> new StringJoiner(" | ") ,

```



(j, r) -> j.add(r.toString()),

*This lambda biconsumer adds  
a new string to the joiner*

```
StringJoiner::merge,
StringJoiner::toString)); ...
```

(j, r) is equivalent to (StringJoiner j, SearchResults.Result r)

# Implementing Custom Non-Concurrent Collectors (Part 1)

- The SearchResults.toString() method uses Collector.of() to format results

```
public String toString() {
 ...
 mList.stream()
 .collect(Collector.of(() -> new StringJoiner(" | ") ,
 (j, r) -> j.add(r.toString())) ,
 StringJoiner::merge,
 StringJoiner::toString) ;
 ...
}
```



(j, r) -> j.add(r.toString()) ,

*Combine two string joiners*

**StringJoiner::merge ,  
StringJoiner::toString) ; ...**

This combiner is only used for parallel streams

# Implementing Custom Non-Concurrent Collectors (Part 1)

- The SearchResults.toString() method uses Collector.of() to format results

```
public String toString() {
 ...
 mList.stream()
 .collect(Collector.of(() -> new StringJoiner(" | ") ,

```

```
(j, r) -> j.add(r.toString()) ,
```

*This finisher converts a string joiner to a string*

```
StringJoiner::merge ,
StringJoiner::toString)) ; ...
```



# Implementing Custom Non-Concurrent Collectors (Part 1)

- The SearchResults.toString() method uses Collector.of() to format results

```
public String toString() {
 ...
 mList.stream()
 .collect(Collector.of(() -> new StringJoiner(" | ") ,

```

*Only four params are passed to of() since Characteristics... is an optional parameter!*

```
(j, r) -> j.add(r.toString()) ,
```

979|1025|1219|1259|  
1278|1300|1351|1370|1835|  
1875|1899|1939|2266|2295]  
Word "Ti" matched at index [237|994|1272|1294|1364|1850|  
1860|1912|1915|1952|1955|  
2299]  
Word "La" matched at index [234|417|658|886|991|1207|  
1247|1269|1291|1339|1361|  
1742|1847|1863|1909|1949|  
2161|2254|2276|2283]...  
Ending SimpleSearchStream"/>

Starting SimpleSearchStream  
Word "Re" matched at index [131|141|151|202|212|222|  
979|1025|1219|1259|  
1278|1300|1351|1370|1835|  
1875|1899|1939|2266|2295]  
Word "Ti" matched at index [237|994|1272|1294|1364|1850|  
1860|1912|1915|1952|1955|  
2299]  
Word "La" matched at index [234|417|658|886|991|1207|  
1247|1269|1291|1339|1361|  
1742|1847|1863|1909|1949|  
2161|2254|2276|2283]...  
Ending SimpleSearchStream

**StringJoiner::merge ,  
StringJoiner::toString)) ; ...**

---

# Implementing Custom Non-Concurrent Collectors (Part 2)

# Implementing Custom Non-Concurrent Collectors (Part 2)

- The WordSearcher.toDownstreamCollector() also uses Collector.of()

```
static Collector<SearchResults, List<SearchResults.Result>,
 List<SearchResults.Result>>
toDownstreamCollector() {
 return Collector.of
 (ArrayList::new,
 (rl, sr) -> rl.addAll(sr.getResultList()) ,
 (left, right) -> {
 left.addAll(right);
 return left;
 });
}
```

See earlier lesson on "Java Streams: Visualizing WordSearcher.printSuffixSlice()"

# Implementing Custom Non-Concurrent Collectors (Part 2)

- The WordSearcher.toDownstreamCollector() also uses Collector.of()

```
static Collector<SearchResults, List<SearchResults.Result>,
 List<SearchResults.Result>>
{
 return Collector.of
 (ArrayList::new,
 toDownstreamCollector() {
 This factory method creates a downstream
 collector that merges results lists together
 (rl, sr) -> rl.addAll(sr.getResultList()) ,
 (left, right) -> {
 left.addAll(right);
 return left;
 });
 }
}
```

See [SimpleSearchStream/src/main/java/search/WordSearcher.java](#)

# Implementing Custom Non-Concurrent Collectors (Part 2)

- The WordSearcher.toDownstreamCollector() also uses Collector.of()

```
static Collector<SearchResults, List<SearchResults.Result>,
 List<SearchResults.Result>>
toDownstreamCollector() {
 return Collector.of
 (ArrayList::new,
 (rl, sr) -> rl.addAll(sr.getResultList()) ,
 (left, right) -> {
 left.addAll(right);
 return left;
 }) ;
}
```

*Factory method creates a new collector via the four-param of() method version*

# Implementing Custom Non-Concurrent Collectors (Part 2)

- The WordSearcher.toDownstreamCollector() also uses Collector.of()

```
static Collector<SearchResults, List<SearchResults.Result>,
 List<SearchResults.Result>>
toDownstreamCollector() {
 return Collector.of(
 (ArrayList::new, ——————Make a mutable results list
container from an array list
 (rl, sr) -> rl.addAll(sr.getResultList()) ,
 (left, right) -> {
 left.addAll(right);
 return left;
 });
}
```

# Implementing Custom Non-Concurrent Collectors (Part 2)

- The WordSearcher.toDownstreamCollector() also uses Collector.of()

```
static Collector<SearchResults, List<SearchResults.Result>,
 List<SearchResults.Result>>
toDownstreamCollector() {
 return Collector.of
 (ArrayList::new,
 (rl, sr) -> rl.addAll(sr.getResultList()) ,
 (left, right) -> {
 left.addAll(right);
 return left;
 });
}
```

*Accumulate all result objects from a SearchResults object into the results list*

# Implementing Custom Non-Concurrent Collectors (Part 2)

- The WordSearcher.toDownstreamCollector() also uses Collector.of()

```
static Collector<SearchResults, List<SearchResults.Result>,
 List<SearchResults.Result>>
toDownstreamCollector() {
 return Collector.of
 (ArrayList::new,
 (rl, sr) -> rl.addAll(sr.getResultList()),
 (left, right) -> {
 left.addAll(right);
 return left;
 });
}
```

*Merge two results lists into a single results list*

This combiner is only used for parallel streams

# Implementing Custom Non-Concurrent Collectors (Part 2)

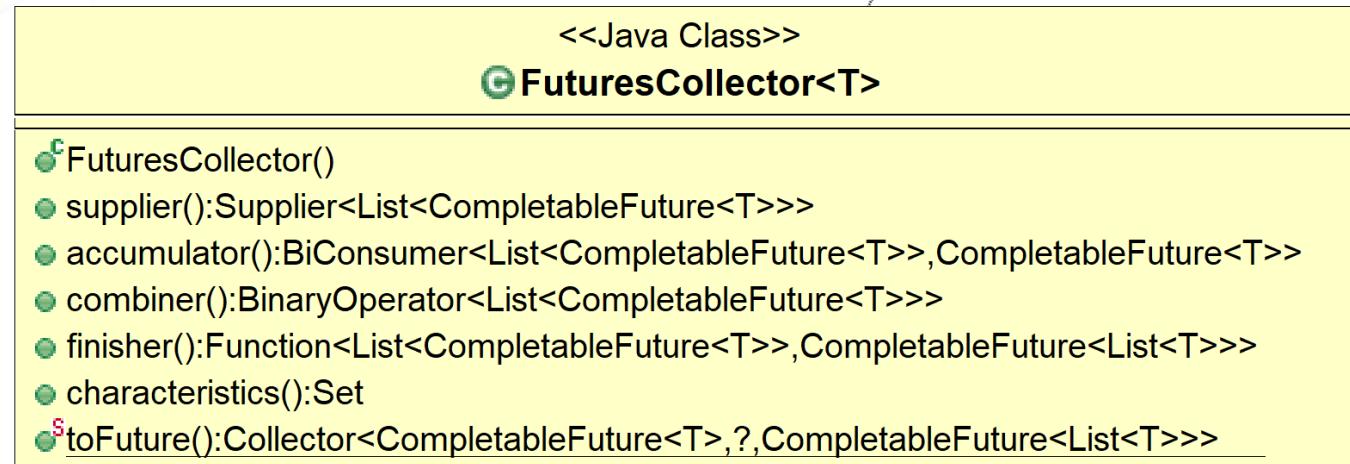
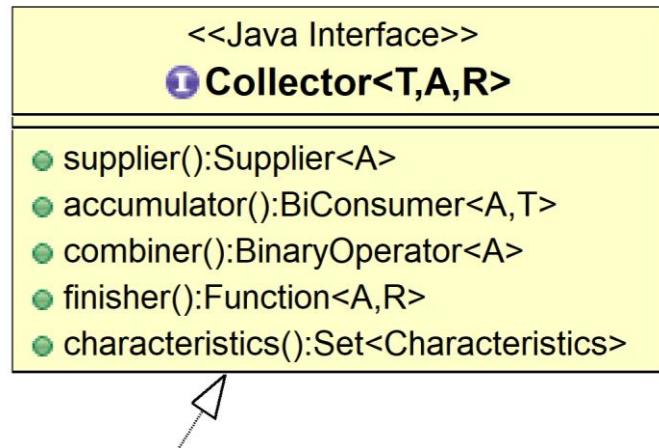
- The WordSearcher.toDownstreamCollector() also uses Collector.of()

```
static Collector<SearchResults, List<SearchResults.Result>,
 List<SearchResults.Result>>
toDownstreamCollector() {
 return Collector.of(
 ArrayList::new,
 (rl, sr) -> rl.addAll(sr.getResultList()),
 (left, right) -> {
 left.addAll(right);
 return left;
 });
}
```

*Only three params are passed to of() since Characteristics... is an optional parameter!*

# Implementing Custom Non-Concurrent Collectors (Part 2)

- Complex custom collectors should implement the Collector interface instead of using Collector.of()



See [Java8/ex19/src/main/java/utils/FuturesCollector.java](https://github.com/Java8/ex19/blob/main/src/main/java/utils/FuturesCollector.java)

# Implementing Custom Non-Concurrent Collectors (Part 2)

- More information on implementing custom collectors is available online

The screenshot shows a video player interface. At the top left is the logo for "iDays GÖTEBORG". Below it, the title "STREAMS IN JAVA 8 (PART 02/02): REDUCE VS COLLEC" is displayed, along with the subtitle "BREV. 1 / DAY 2 / 9 MARCH 2016 / 15:30-16:15" and the name "Angelika Langer, Angelika Langer Training & Consulting". To the right of the title is a large code block titled "example – accumulator". The code implements a collector for strings:

```
public void accumulate(String nextLine) {
 if (nextLine != null) {
 int indexOfLastEntry = result.size()-1;
 if (indexOfLastEntry < 0) {
 result.add(indexOfLastEntry+1,nextLine);
 } else {
 String current = result.get(indexOfLastEntry);
 if (current.length() == 0)
 result.add(indexOfLastEntry+1, nextLine);
 else {
 char endChar = current.charAt(current.length()-1);
 if (endChar == '\\')
 result.set(indexOfLastEntry, current.substring
 (0, current.length()-1) + nextLine);
 else
 result.add(indexOfLastEntry+1, nextLine);
 } }
}
```

At the bottom of the video player, there is a copyright notice: "© Copyright 2003-2016 by Angelika Langer & Klaus Kretz. All Rights Reserved. http://angelikalanger.com/ last update: 3/5/2016, 15:05". On the right side of the video player, the text "reduce/collect (77)" is visible. The video player has a red progress bar at the bottom, indicating the video is at 31:28 of 51:11. It also includes standard video controls like play/pause, volume, and full screen.

See [www.youtube.com/watch?v=H7VbRz9aj7c](https://www.youtube.com/watch?v=H7VbRz9aj7c)

---

End of Learn How to  
Implement Custom Non-  
Concurrent Collectors

# Learn How to Avoid Common Java Streams Programming Mistakes

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

Professor of Computer Science

Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Lesson

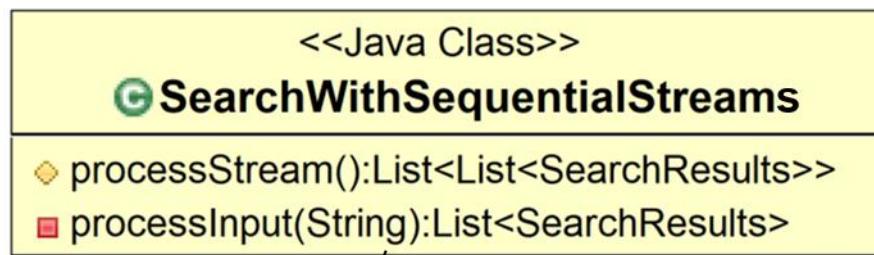
---

- Know how to apply sequential streams to the SearchStreamGang program
- Recognize how a Spliterator is used in SearchWithSequentialStreams
- Understand the pros & cons of the SearchWithSequentialStreams class
- Learn how to avoid common streams programming mistakes



## Learning Objectives in this Part of the Lesson

- Know how to apply sequential streams to the SearchStreamGang program
  - Recognize how a Spliterator is used in SearchWithSequentialStreams
  - Understand the pros & cons of the SearchWithSequentialStreams class
  - Learn how to avoid common streams programming mistakes



*We discuss several examples in this lesson, including `SearchWithSequentialStreams`.*

See [streamgangs/SearchWithSequentialStreams.java](#)

---

# Avoiding Common Streams Programming Mistakes

# Avoiding Common Streams Programming Mistakes

- Don't forget the terminal operation!

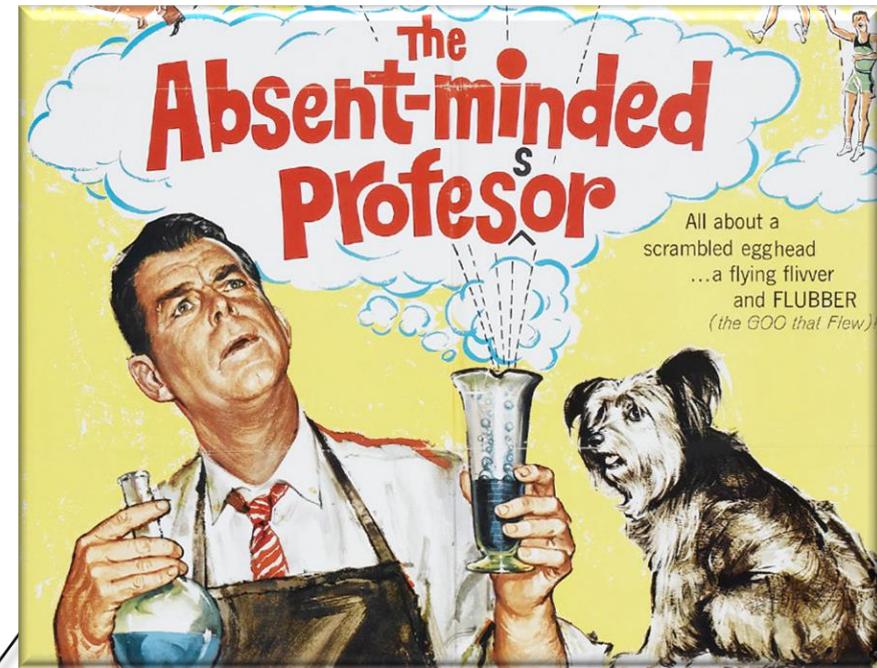
```
List<CharSequence> input =
 getInput();
```

```
Stream<List<SearchResults>> input
 .stream()

.map(this::processInput);
```



*This is an all-to-common beginner mistake..*

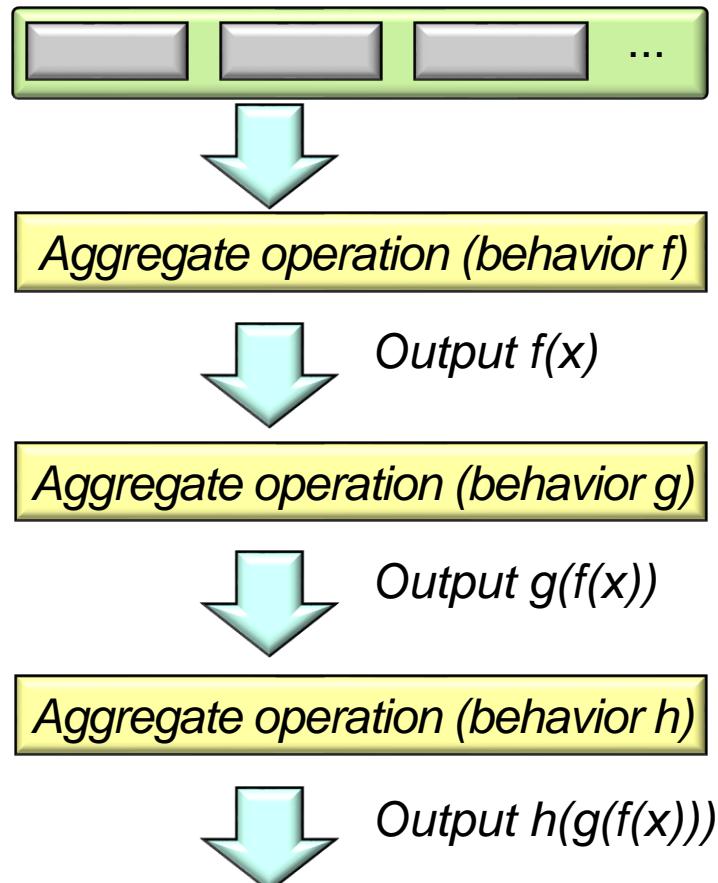


See [streamgangs/SearchWithSequentialStreams.java](#)

# Avoiding Common Streams Programming Mistakes

- Only traverse a stream once

One  
life  
One  
Chance



# Avoiding Common Streams Programming Mistakes

- Only traverse a stream once

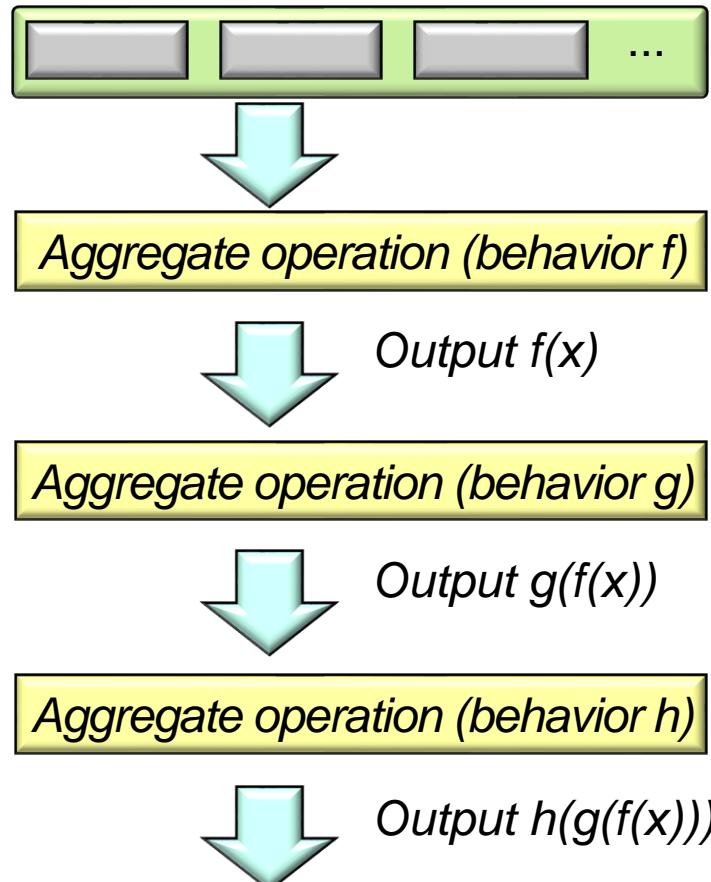
```
List<CharSequence> input =
 getInput();
```

```
Stream<List<SearchResults>> s = input
 .stream()
```

```
 .map(this::processInput);
```

```
s.forEach(System.out::println);
s.forEach(System.out::println);
```

*Duplicate calls are invalid!*



# Avoiding Common Streams Programming Mistakes

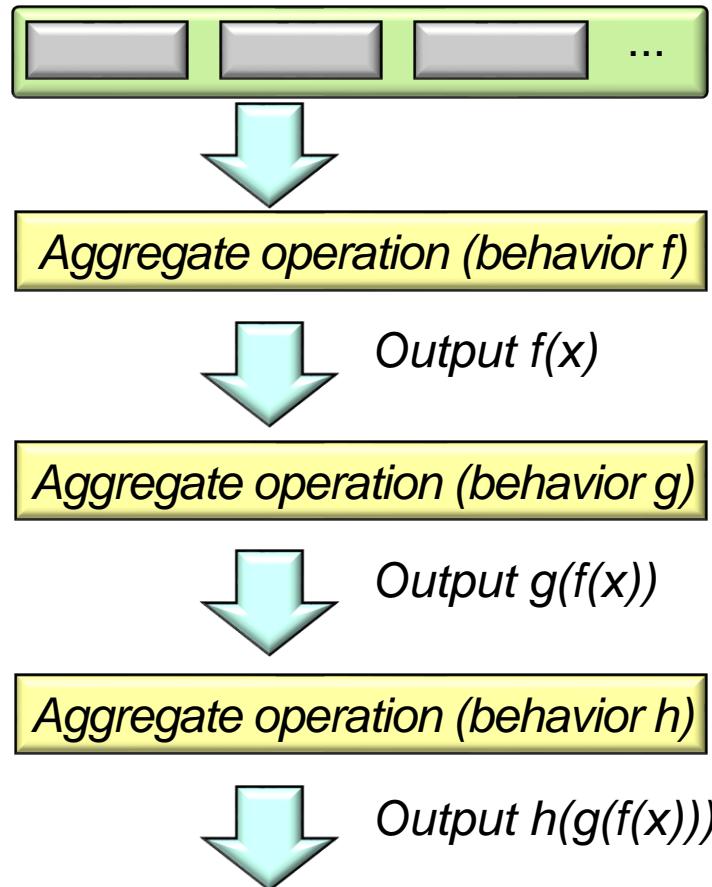
- Only traverse a stream once

```
List<CharSequence> input =
 getInput();
```

```
Stream<List<SearchResults>> s = input
 .stream()
 .map(this::processInput);

s.forEach(System.out::println);
s.forEach(System.out::println);
```

Throws `java.lang.IllegalStateException`

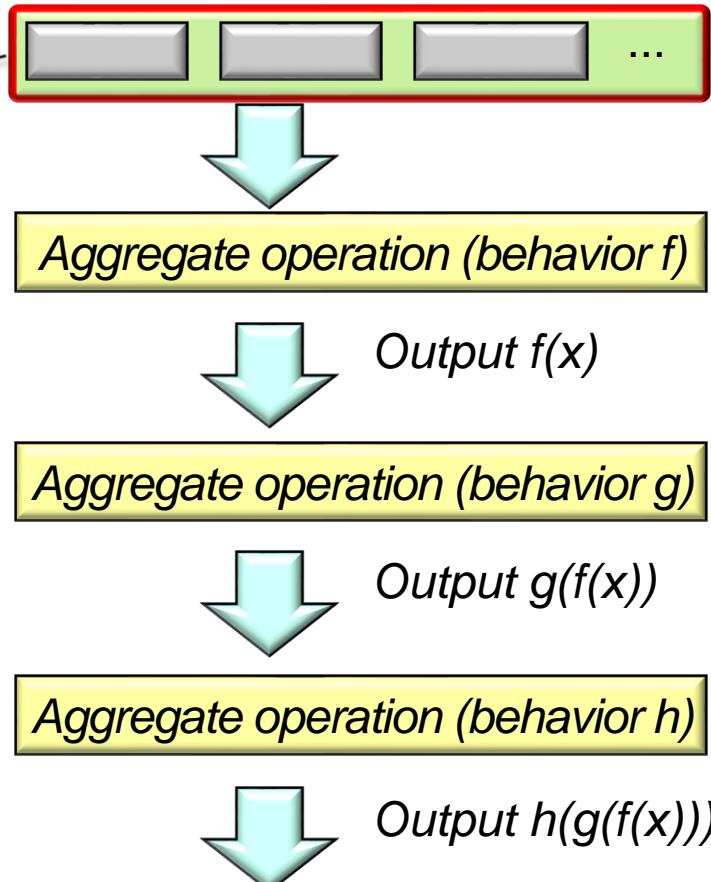


See [docs.oracle.com/javase/8/docs/api/java/lang/IllegalStateException.html](https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalStateException.html)

# Avoiding Common Streams Programming Mistakes

- Only traverse a stream once

*To traverse a stream again you need to get a new stream from the data source*



# Avoiding Common Streams Programming Mistakes

---

- Don't modify the backing collection of a stream

```
List<Integer> list = IntStream
 .range(0, 10)
 .boxed()
 .collect(toList());
```

```
list
 .stream()
 .peek(list::remove)
 .forEach(System.out::println);
```

# Avoiding Common Streams Programming Mistakes

- Don't modify the backing collection of a stream

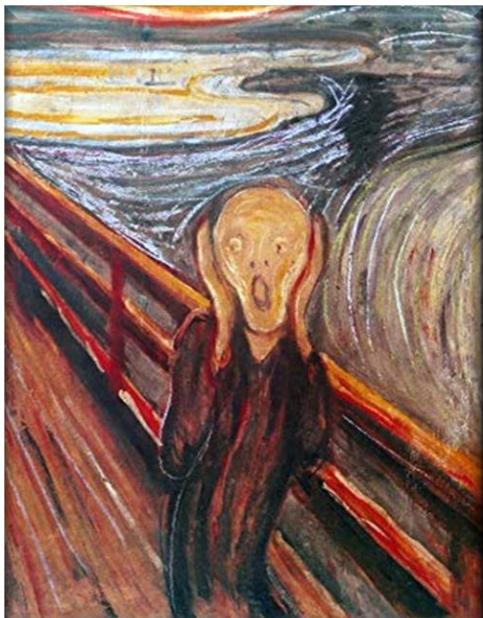
```
List<Integer> list = IntStream
 .range(0, 10)
 .boxed()
 .collect(toList());
```

*Create a list of ten integers in range 0..9*

```
list
 .stream()
 .peek(list::remove)
 .forEach(System.out::println);
```

# Avoiding Common Streams Programming Mistakes

- Don't modify the backing collection of a stream



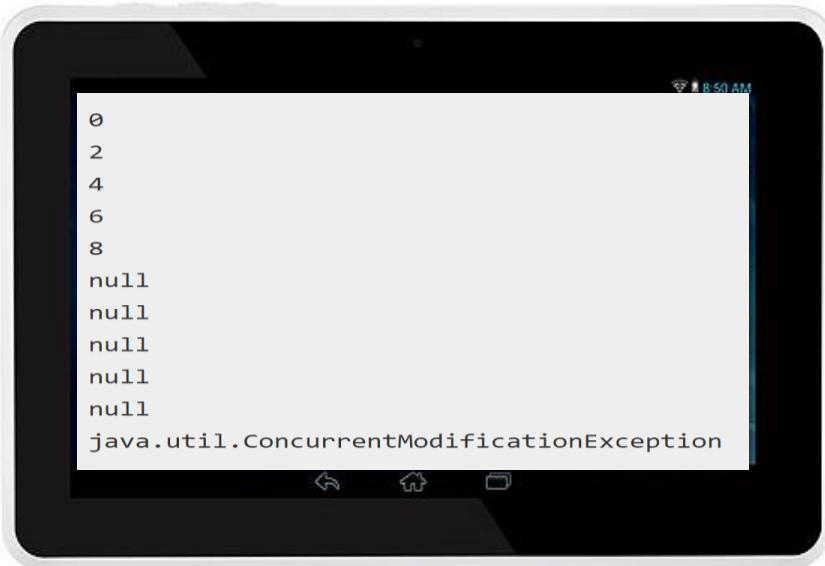
```
List<Integer> list = IntStream
 .range(0, 10)
 .boxed()
 .collect(toList());
```

```
list
 .stream()
 .peek(list::remove)
 .forEach(System.out::println);
```

*If a non-concurrent collection is modified while it's being operated on the results will be chao & insanity!!*

# Avoiding Common Streams Programming Mistakes

- Don't modify the backing collection of a stream

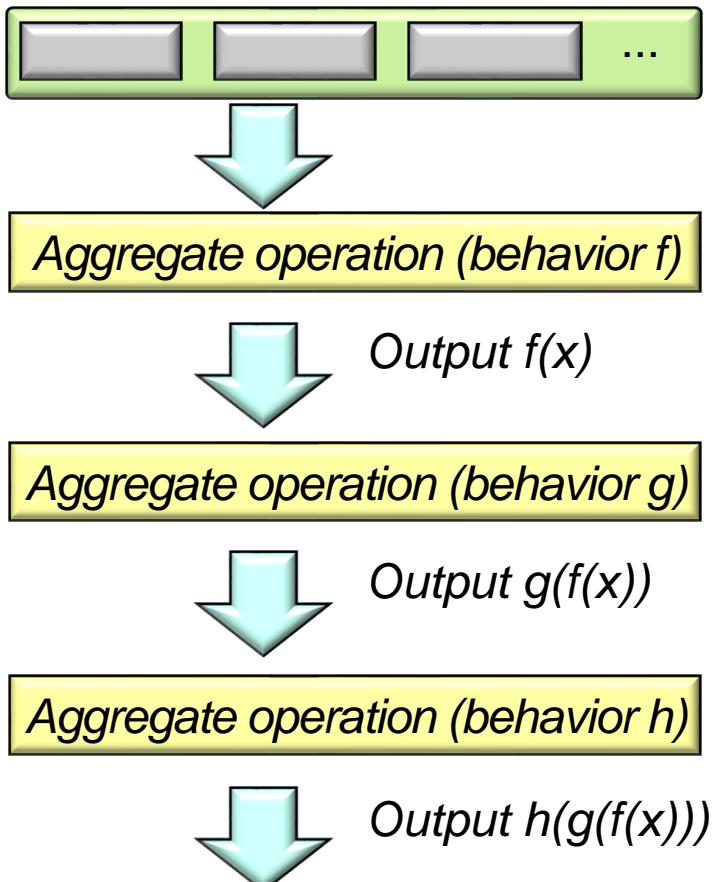


```
list
 .stream()
 .peek(list::remove)
 .forEach(System.out::println);
```

*Modifying a list while it's been iterated/  
splintered through will yield weird results!*

# Avoiding Common Streams Programming Mistakes

- Remember that a stream holds no non-transient storage

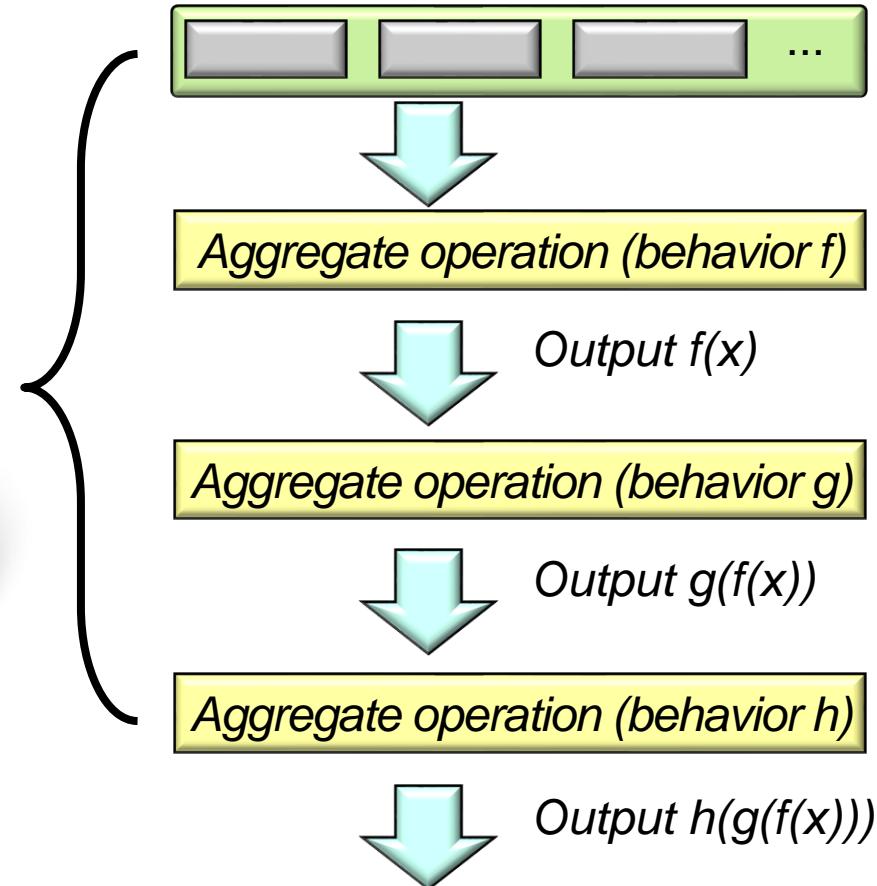


# Avoiding Common Streams Programming Mistakes

- Remember that a stream holds no non-transient storage



*Apps are responsible for persisting any data that must be preserved*



---

# End of Learn How to Avoid Common Java Streams Programming Mistakes